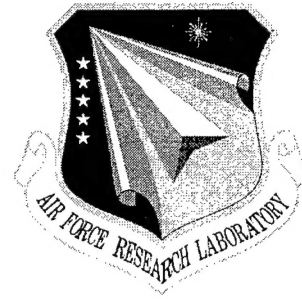AFRL-IF-RS-TR-1998-153
Final Technical Report
July 1998

SIMS II: SINGLE INTERFACE TO MULTIPLE SOURCES OF INCOMPLETE INFORMATION

USC Information Sciences Institute

Sponsored by
Defense Advanced Research Projects Agency
DARPA Order No. B396

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

**AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE
ROME RESEARCH SITE
ROME, NEW YORK**

19980911 006

This report has been reviewed by the Air Force Research Laboratory, Information Directorate, Public Affairs Office (IFOIPA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

AFRL-IF-RS-TR-1998-0000 has been reviewed and is approved for publication.

APPROVED:

RAYMOND A. LIUZZI
Project Engineer

FOR THE DIRECTOR:

NORTHRUP FOWLER III, Technical Advisor
Information Technology Division
Information Directorate

If your address has changed or if you wish to be removed from the Air Force Research Laboratory Rome Research Site mailing list, or if the addressee is no longer employed by your organization, please notify AFRL/IFTB, 525 Brooks Rd, Rome, NY 13441-4505. This will assist us in maintaining a current mailing list.

Do not return copies of this report unless contractual obligations or notices on a specific document require that it be returned.

SIMS II: SINGLE INTERFACE TO MULTIPLE SOURCES OF INCOMPLETE INFORMATION

Yigal Avens

Contractor: USC Information Sciences Institute
Contract Number: F30602-94-C-0210
Effective Date of Contract: 31 August 1994
Contract Expiration Date: 31 December 1997
Program Code Number: 62301E
Short Title of Work: SIMS II: Single Interface to Multiple Sources of Incomplete Information
Period of Work Covered: Aug 94 - Dec 97

Principal Investigator: Yigal Avens
Phone: (310) 822-1511
AFRL Project Engineer: Raymond A. Liuzzi
Phone: (315) 330-3577

Approved for public release; distribution unlimited.

REPORT DOCUMENTATION PAGE

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	Jul 98	Final Aug 94 - Dec 97

4. TITLE AND SUBTITLE

SIMS II: SINGLE INTERFACE TO MULTIPLE SOURCES OF INCOMPLETE INFORMATION

6. AUTHOR(S)

Yigal Arens

5. FUNDING NUMBERS

C   - F30602-94-C-0210
PE - 62301E
PR - B396
TA - 00
WU - 01

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

USC Information Sciences Institute
4676 Admiralty Way
Marina Del Rey, CA 90292-6695

8. PERFORMING ORGANIZATION REPORT NUMBER

N/A

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Defense Advanced Research Projects Agency
3701 North Fairfax Drive
Arlington, VA 22203-1714

AFRL/IFTB
525 Brooks Rd.
R ome, NY 13441-4505

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

AFRL-IF-RS-TR-1998-153

11. SUPPLEMENTARY NOTES

AFRL Project Engineer: Raymond A. Liuzzi, IFTB, 315-330-3577

12a. DISTRIBUTION AVAILABILITY STATEMENT

Approved for public release; distribution unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)

With the current explosion of data, retrieving and integrating information from various sources is a critical problem. This report describes work performed at USC/ISI, aimed at developing a general and extensible approach to this problem, with attention paid also to cases where knowledge of the ultimate sources is incomplete or inaccurate. The SIMS approach exploits a semantic model of a problem domain to integrate the information from various sources -- databases, knowledge bases and more. The domain and the information sources are modeled. Queries submitted to SIMS are mapped into a set of queries to individual information sources. The data obtained is then returned to the user. SIMS utilizes techniques from the areas of knowledge representation, planning, learning, and data mining.

14. SUBJECT TERMS

Information Integration, Databases, Data Mining, Planning, Information Management, Artificial Intelligence

15. NUMBER OF PAGES

124

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

Contents

Abstract

With the current explosion of data, retrieving and integrating information from various sources is a critical problem. This report describes work performed at USC/ISI, aimed at developing a general and extensible approach to this problem, with attention paid also to cases where knowledge of the ultimate sources is incomplete or inaccurate. The SIMS approach exploits a semantic model of a problem domain to integrate the information from various sources — databases and knowledge bases. The domain and the information sources are modeled. Queries submitted to SIMS are mapped into a set of queries to individual information sources. The set of queries is then further optimized, using knowledge about the domain and the information sources. The data obtained is then returned to the user. SIMS utilizes techniques from the areas of knowledge representation, planning, learning, and data mining.

Work on the SIMS system was funded also by an earlier DARPA contract. After providing only a brief review of the basic SIMS system, this report will concentrate on new developments funded exclusively by this contract. Further background can be found in earlier publications, listed in the references.

This report is structured as follows. Chapter 1 provides a brief overview of SIMS. The following three chapters provide descriptions of the major revisions and improvements to SIMS planning done during the period of this contract. Chapter 2 describes modifications to SIMS' planner that made it particularly suitable for the domain of information gathering. Chapter 3 describes how planning efficiency was improved by pre-processing source descriptions. Chapter 4 describes Planning By Rewriting (PBR), a new approach to planning developed late in the course of the project. The final two chapters address issues that arise due to the incompleteness of the system's knowledge of sources, or variations in the makeup of the system that we have no control over. Chapter 5 describes our approach to replanning when sources that the system believes to be available are unexpectedly inaccessible, and Chapter 6 describes our use of data mining techniques to help come up with models and other information the system needs, by inspecting databases directly. We conclude with an appendix, a tutorial describing how to set up a SIMS system to access new information sources.

Chapter 1

Brief Summary of SIMS

1 Introduction

Most tasks performed by users of complex information systems involve interaction with multiple information sources.[1] The SIMS approach to this integration problem has been based largely on research in Artificial Intelligence; primarily in the areas of knowledge representation, planning, and machine learning. A model of the application domain is created, using a knowledge representation system to establish a fixed vocabulary for describing objects in the domain, their attributes and relationships among them. Using this vocabulary, a description is created for each information source. Each description indicates the data-model used by the source, the query language, network location, size estimates, etc., and describes the contents of its fields in relation to the domain model. SIMS' descriptions of different information sources are independent of each other, greatly easing the process of extending the system. Some of the modeling is aided by source analysis software developed as part of the SIMS effort.

Queries to SIMS are written in a high-level language (Loom or a subset of SQL) using the terminology of the domain model — independent of the specifics of the information sources. Queries need not contain information indicating which sources are relevant to their execution or where they are located. Queries do not need to state how information present in different sources should be joined or otherwise combined or manipulated.

SIMS uses a planner to determine how to identify and combine the data necessary to process a query. In a pre-processing stage, all data sources possibly relevant to the query are identified. The planner then selects a set of sources that contain the queried information and generates an initial plan for the query. This plan is repeatedly refined and optimized until it meets given performance criteria. The plan itself includes, naturally, sub-queries to appropriate information sources, specification of locations for processing intermediate data, and parallel branches when appropriate. The SIMS system then executes the plan. The plan's execution is monitored and replanning is initiated if its performance meets with difficulties such as unexpectedly unavailable sources. It is also possible for the plan to include explicit replanning steps, after reaching a state where more is known about the circumstances of plan execution.

Changes to information sources are handled by changing source descriptions only. The changes will automatically be considered by the SIMS planner in producing future plans that utilize information from the modified sources. This greatly facilitates extensibility.

1.1 Architecture

A visual representation of the components of SIMS is provided in Figure 1.1.

SIMS addresses the problems that arise when one tries to provide a user familiar only with the general domain with access to a system composed of numerous separate data- and knowledge-bases.

---

[1] By the term *information source* we refer to any system from which information can be obtained. See Section 1.2 for a listing.
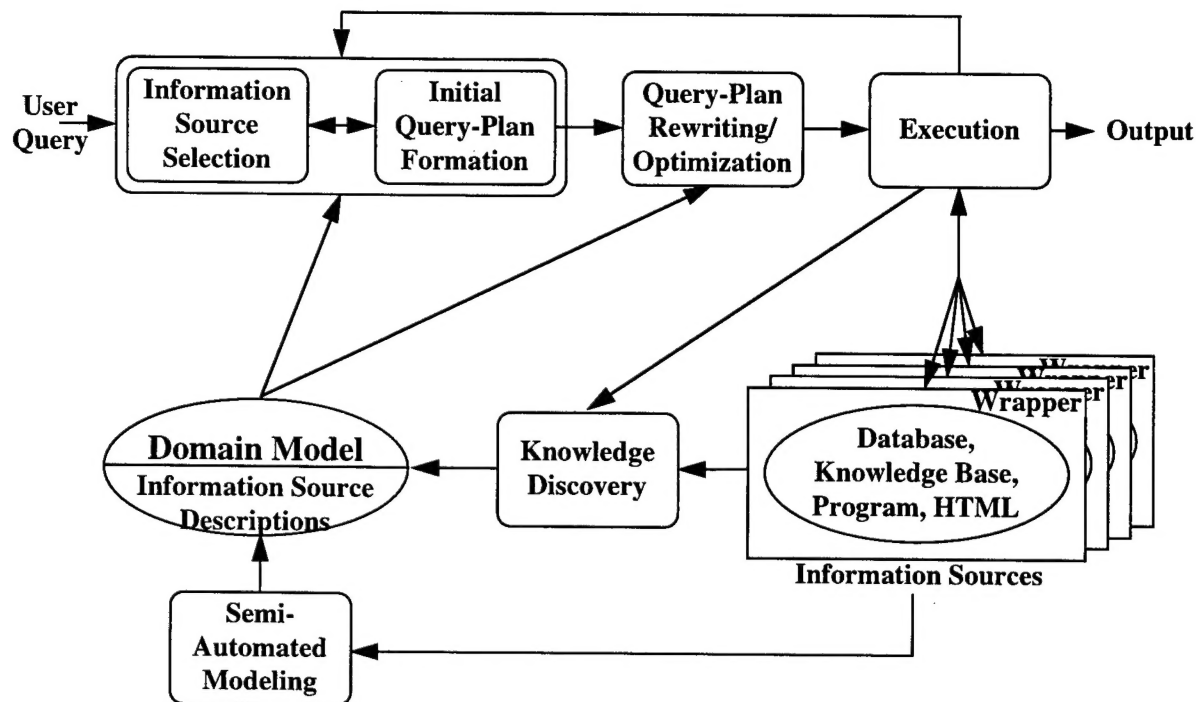
Figure 1.1: SIMS Overview Diagram.

Specifically, SIMS does the following:

- **Modeling:** It provides a consistent way of describing information sources to the system, so that data in them is accessible to it.

- **Information Source Selection:** Given a query, it

  - Determines which classes of information will be relevant to answering the query.

  - Quickly, using some information generated during an earlier preprocessing stage, generates a list of all combinations of sources that contain all information required for a query.

- **Initial Query-Plan Formation:** It creates an initial plan, a sequence of subqueries and other forms of data-manipulation that when executed will yield the desired information. This initial plan does not necessarily satisfy any optimization requirements.

- **Query-Plan Rewriting/Optimization:** By successively applying rewriting rules that preserve the correctness of the plan, it gradually improves the plans efficiency. This process continues until no further rewriting is possibly, or until the allotted time runs out.

- **Semi-Automated Modeling:** By querying databases and other information sources and analyzing the returned information, it discovers semantic rules characterizing their contents. This learned knowledge is used to help create SIMS' information source descriptions.

- **Execution:** It executes the reformulated query plan; establishing network connections with the appropriate information sources, transmitting queries to them and obtaining the results for further processing. During the execution process SIMS may detect that certain information sources are not available, or respond erroneously. In such cases, the relevant portion of the query plan will be replanned. In addition, certain plans will contain steps that require replanning some plan branch some time into the execution phase, after more information is known.

6

```
(db-retrieve (?depth)
   (:and (port ?port)
         (port.name ?port "SAN-DIEGO")
         (port.depth ?port ?depth)))
```

Figure 1.2: Example SIMS/Loom Query

Each information source is accessed through a *wrapper*, a module that can translate from a description of a set of data in SIMS' internal representation language (Loom) into a query for that data that is then submitted to the source. The wrapper also handles communication with the information source and takes the data returned by it and sends it on to SIMS in the form SIMS expects.

1.2 Information Sources Supported

In order for SIMS to support an information source it must have a description of the source, and there must exist a wrapper for that type of source. While each information source needs to be described individually, only one wrapper is required for any type of information source.

Wrappers for Loom knowledge bases and MUMPS-based network databases have been written for SIMS. In addition, through an "ODBC wrapper" SIMS uses ODBC to interact with all ODBC-enabled databases. This includes Oracle, Sybase, Informix, Ingres, and many others. To add a new database of any of these types requires, therefore, only to create an information source description for it. In order to add an information source of a new type one would have to obtain, or write, a new wrapper for it as well. We are currently working on wrappers for certain object oriented databases. We also have an ongoing associated effort (Ariadne) that includes work on semi-automatic generation of wrappers for HTML pages.

1.3 Loom

This subsection is provided for readers who may not be familiar with the knowledge representation system underlying SIMS.

Loom serves as the knowledge representation system SIMS uses to describe the domain model and the contents of the information sources, as well as serving as an information source in its own right. It provides both a language and an environment for constructing intelligent applications. Loom combines features of both frame-based and semantic network languages, and provides some reasoning facilities. As a knowledge representation language it is a descendent of the KL-ONE [12] system.

The heart of Loom is a powerful knowledge representation system, which is used to provide deductive support for the declarative portion of the Loom language. Declarative knowledge in Loom consists of definitions, rules, facts, and default rules. A deductive engine called a *classifier* utilizes forward-chaining, semantic unification and object-oriented truth maintenance technologies in order to compile the declarative knowledge into a network designed to efficiently support on-line deductive query processing. For a detailed description of Loom see [49, 50].

To illustrate both Loom and the form of SIMS' queries, consider Figure 1.2, which contains a very simple semantic query to SIMS. This query requests the value of the depth of the San Diego port. The three subclauses of the query specify, respectively, that the variable ?port describes a member of the model class port, that the relation port.name holds between the value of ?port and the string SAN-DIEGO, and that the relation port.depth holds between the value of ?port and the value of the variable ?depth. The semantic query specifies that the value of the variable ?depth be returned. A query to SIMS need not necessarily correspond to a single database query, since there may not exist one database that contains all the information requested.

Chapter 2

Building a Planner for Information Gathering

1 Introduction

This chapter examines the issues in applying a general-purpose planner to the information gathering task. Information gathering involves locating and integrating information to answer queries from a set of heterogeneous and distributed information sources. An information gathering plan includes the source for the information, the specific operations that are to be performed on the data, and the order in which the operations are to be performed. The general problem is quite hard since there are a very large number of ways a query can be processed. The choice of plans is critical since the cost of executing different plans for the same query can range from a few milliseconds to a few years or longer.

Consider the example query, shown in Table 2.1, from a transportation domain, which requests the names of all seaports that have channels that can accommodate small tankers. A plan to answer this query is shown in Figure 2.1. In this example, the plan partitions the given query such that in parallel the seaport information is retrieved from the SEAPORT information source and the tanker information is retrieved from the ASSETS information source. The results of those two queries are then joined in the local system. The plan then retrieves the information from the COUNTRY source and compares it to the intermediate results of the other subplan. Once the system generates the final set of data, it is sent to the output.

```
(retrieve (?port-name)
   (:and (seaport ?sport)
         (port-name ?sport ?port-name)
         (country-name ?sport "Saudi Arabia"
         (channel-of ?sport ?channel)
         (channel-depth ?channel ?depth)
         (transport-ship ?ship)
         (vehicle-type-name ?ship "Small Tanker")
         (max-draft ?ship ?draft)
         (< ?draft ?depth)))
```

Table 2.1: An Example Query

The information gathering task has a number of characteristics that make it an interesting planning problem. First, the basic problem is to generate a sequence of actions to efficiently retrieve the requested set of information. Unlike much of the previous work on planning, a *satisficing* solution is not sufficient since the quality of the final plan is an important consideration. Second, the search space is large, but there are fewer interactions than there are in domains such as the blocksworld, so it is possible to search a large space of possible plans. Third, the plans produced in this domain can be directly executed, which makes it possible to consider issues of planning and execution without the use of a simulator. Finally, and most
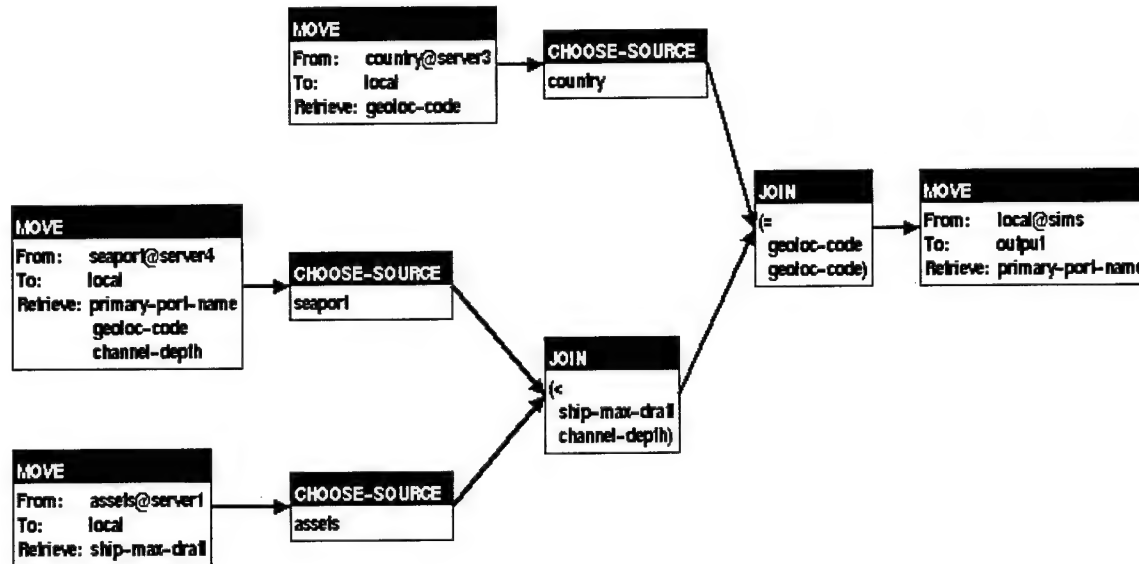
Figure 2.1: An information gathering plan

importantly, this is a real problem.

This problem is similar to the problem of generating a query access plan for a database in that both require producing a sequence of actions to produce the requested data. However, it differs from query access planning in several ways. The first difference is that query access planning only requires generating a plan for how to process the data, while information gathering also requires identifying the relevant sources for use in producing the requested information. The second difference is that query access planning is also concerned with the low-level implementation of the query processing to minimize the processing time and amount of disk I/O. In our formulation of the information gathering task, we are only concerned with determining how the queries should be partitioned into subqueries to the remote sources and we leave it up to the individual systems to implement the subqueries efficiently.

In earlier papers [38, 39] we addressed various issues related to planning for information gathering, such as simultaneous actions, integrating planning and execution, replanning, and sensing. This chapter focuses on the issues that arise in taking a general-purpose planner and using it to construct a planner for the information gathering task. In order to provide insight into some of the design decisions we include some discussion of how this planner has evolved over the last few years.

The chapter is organized as follows: First, we describe the underlying general-purpose planner and identify both the important features and additional functionality that had to be added. Next, we describe how we represent the information gathering task as a planning problem and identify some of the important design principles. Given this representation, we then describe the control knowledge and evaluation function used to efficiently generate high-quality information gathering plans. In order to demonstrate that the system can efficiently generate plans, we provide some empirical results that compare the planning time to the execution time. Then we compare this work to work on query access planning as well as other related work on information gathering. Finally, we identify some of the critical planning research problems, and then describe some of our plans for future work.

2 The Basic Planner

The first prototype of the overall system [8, 7] was built using the Prodigy 2.0 planner [53]. Prodigy was chosen because it has a expressive operator representation language and a rich language for expressing control knowledge. This initial version of the system produced a satisficing solution in that it searched the space using a set of heuristics that produced a reasonably good solution. Since an important aspect of this problem is plan quality, generating high quality plans requires exploiting the fact that remote sources can be accessed

in parallel. To do this required explicitly representing the potential parallelism in the plan. Since Prodigy produces totally-ordered plans, the plans produced by Prodigy were converted into a parallel-execution plans using the algorithm of Veloso [70], which converts a totally-ordered plan into a partially ordered plan.

In the next version of the system we switched to UCPOP 2.0 [10] because we needed a partial-order planning capability. The initial version in Prodigy returned the first plan produced using the set of heuristics, but as the plans became more complex it became clear that this was insufficient. In order to consider the tradeoffs in different ways of processing a query, we needed to have the planner construct alternative plans and evaluate the alternatives to find a high-quality plan. To do this using a total-order planner would be costly since the final result is a partially-ordered plan and a number of different totally-ordered plans can all map to the same partially-ordered plan. To avoid considering redundant plans and to evaluate the partially-constructed plans, a partial-order planner is better suited to this particular task. In addition to providing a partial-order planning capability, UCPOP provides a number of important features of Prodigy, including an expressive operator language, a separate control language, and the capability of defining preconditions using *functional predicates* (described below).

The expressive operator language is necessary for defining the operators in the information gathering domain. The language features used in the current version of the domain are conjunction, disjunction, existential, and universal quantification. The control language was used initially for this domain in both Prodigy and UCPOP, but was later abandoned for efficiency reasons.

The ability to define preconditions using functional predicates is also a critical feature for representing this problem. This feature allows certain predicates to be defined as functions, where given bindings for some of the variables it computes the bindings of the other variables. This capability allows the operators to be defined at an appropriate level of abstraction. The aspect of the information gathering problem that is well suited to a planner is the search through the space of possible operations for selecting the sources and manipulating the data. There are other aspects to this problem such as computing the possible partitions of a query or determining whether a set of data is contained in a particular information source that would be difficult to capture in a planner. The ability to define preconditions using functional predicates turns out to be particularly useful since much of the low-level query processing can be done in Lisp code, which allows the planner to focus on the search at an appropriate level of granularity.

There are several capabilities that are not provided by UCPOP that are required to solve the task. These are the ability to represent and manipulate complex terms and the explicit representation of resources. The representation of complex terms is essential since the specification of a set of data can be quite involved. It includes the specification of the classes of objects, the relevant roles on those objects, constraints on the individual objects as well as constraints between objects, and so on. An example of a complex term is shown in the example query shown in Table 2.1. Terms such as these would be impractical to define as a flat literal with a fixed number of parameters. It would also be difficult to plan for queries if they were represented by a set of flat literals since that would require reasoning about the achievement of several simultaneous goals and existing planners are not particularly well suited to that problem.

In order to execute several actions simultaneously in a partially-ordered plan requires an explicit representation of reusable resources. A partially-ordered plan only allows actions that are unordered with respect to each other to be executed in either order. It does not sanction simultaneous execution. To provide simultaneous execution requires the addition of explicit resource constraints [73, 38]. The advantage of parallel plans are that they can greatly reduce the overall execution time of queries by supporting simultaneous execution. The representation of reusable resources allows the system to explicitly reason about potential resource conflicts and take them into account in order to generate the best plan to solve the problem. An alternative approach to address this problem would be to schedule the operations after the planning is complete; however, this approach would be less efficient since in some cases there will be several different operators for achieving the same goal and the choice of operator, which would be made before the scheduling is performed, could have a significant impact on the schedule.

In order to address the needs of this application, we built a general planner called Sage, which extends the UCPOP planner in several important ways. First, we added support for compound objects, which required extending the matching algorithm. Second, we added an explicit representation of reusable resources and extended the planner to identify possible resource conflicts and refine the plan to eliminate them. This allows the planner to generate plans with simultaneous actions. (See [38] for more detail.)

10

In addition, we added support in Sage for simultaneous and interleaved planning and execution. This is done by tightly integrating the execution into the planning process and letting the planner decide which actions to execute and when to execute them. This allows the planner to replan failed actions, handle asynchronous goals, and interleave the planning and sensing. To support the sensing we added explicit run-time variables [6, 21] which provide a mechanism for the planner to use the results of a sensing action. The support for execution, replanning, and sensing are described in [39] and will not be described further in this chapter. In the remainder of this chapter we will focus on how the information gathering task is represented in Sage and how the planner efficiently generates plans in this domain.

3   Representing the Problem

The problem to be solved in this task is to determine a sequence of actions to efficiently retrieve a requested set of data (i.e., the query). In the initial representation of this problem in Prodigy, we cast as much of this problem as possible as a planning problem [7]. The operators manipulated the individual terms of a query to determine which terms depended on other terms, which in turn determines the order in which different queries would be executed. Once a complete plan graph was constructed, the resulting plan had to then be converted into an actual query plan that could be executed. There were several important limitations of this representation. First, the operators in the resulting plan did not correspond directly to the actual actions that would be executed, which made it difficult to evaluate intermediate plans to more efficiently search the space. Second, much of the work that was being done by the planner involved analyzing the structure of the query, which could be done more efficiently in Lisp code.

In the next version implemented in UCPOP, we redesigned the representation of the problem with two design goals in mind. First, we wanted the operators in the plan to correspond directly to the operators that would actually be executed. This simplified the evaluation of intermediate plans and made it possible to integrate the planning and execution. Second, we wanted the preconditions of these operators to be completely self-contained goals. This simplified and modularized the overall design of the operators and made it possible to extend the domain in new directions such as the integration of sensing into the system. The difficulty in this choice of representation is that it placed a much greater burden on the processing required for each individual operator. There are now more than 10,000 lines of Lisp code used to define the functional predicates used in the operators.

Using this representation, a query is cast as an information goal. Such a goal consists of a description of a set of desired data (i.e., the query) as well as the location where that data is to be sent. The form of this goal is:

    (available <source> <server> <query>).

The <source> is either the name of a remote source, local to indicate the local knowledge base, or output to indicate that the results should be displayed. The <server> is either the name of a server that can provide one or more sources (e.g., an Oracle DBMS), or it is the name of the local information mediator, which is called sims in the example. Finally, the <query> is a description of desired information. An example of an information goal is shown in Table 2.2.

```
(available output sims
   (retrieve (?port-name)
      (:and (seaport ?sport)
            (port-name ?sport ?port-name)
            (channel-of ?sport ?channel)
            (channel-depth ?channel ?depth)
            (transport-ship ?ship)
            (vehicle-type-name ?ship "breakbulk")
            (max-draft ?ship ?draft)
            (< ?draft ?depth))))
```

Table 2.2: An information gathering goal

All of the operators in this domain manipulate these information goals. There are two types of operators: those that correspond to data manipulation operators and those that simply reformulate an information goal

11

into an equivalent goal and are used to select an appropriate set of sources. The data manipulation operators include `move`, `join`, `update`, `assign`, `select`, and `compute`. The reformulation operators include: `choose-source`, `infer-equivalence`, `substitute-definition`, and `decompose`. The reformulation operators are used to rewrite the query expressed in domain terms into queries expressed in terms of the available information sources. The details of the reformulation operators are provided in [9].

Consider the operator shown in Table 2.3, which defines a join performed in the local system. This operator is used to achieve the goal of making some information available in the local knowledge base of the SIMS information mediator. It does this by partitioning the request into two subsets of the requested data, retrieving that information into the local system, and then joining the data together to produce the requested set of data.

```
(define (operator join)
  :parameters (?join-op ?data ?data-a ?data-b)
  :precondition
    (:and (join-partition ?data ?join-op
                          ?data-a ?data-b)
          (available local sims ?data-a)
          (available local sims ?data-b))
  :effect (available local sims ?data))
```

Table 2.3: The join operator

The `join-partition` precondition is defined by a functional predicate that produces the relevant partitions of the requested data. For example, the query described above would be partitioned into two subqueries based on how the information is organized in the underlying information sources. Since information on `transport-ship` and `seaport` is located in different sources, the function would return the partition shown in Table 2.4.

Join Constraint:
```
(< ?draft ?depth)
```

Subquery 1:
```
(retrieve (?port-name ?depth)
  (:and (seaport ?sport)
        (port-name ?sport ?port-name)
        (channel-of ?sport ?channel)
        (channel-depth ?channel ?depth)))
```

Subquery 2:
```
(retrieve (?draft)
  (:and (transport-ship ?ship)
        (vehicle-type-name ?ship "breakbulk")
        (max-draft ?ship ?draft)))
```

Table 2.4: Result of calling Join-Partition on the example query

The functional predicates process queries based on a model of the domain and models of the contents of the information sources. This information comprises the static part of the initial state information. To organize this information and access it efficiently, these models are stored in a knowledge representation system called Loom [50]. The information is then accessed directly through the functional predicates, which make direct calls to Loom.

The dynamic part of the initial state information is comprised of literals that define the available information sources (e.g., databases) and the servers (e.g., an Oracle DBMS) they are running on. The example shown in Table 2.5 defines four databases running on three different servers. The first two lines define two Oracle databases running on a single Oracle server. The second two lines define two Loom knowledge bases running on different servers.

12

```
((source-available assets server1)
 (source-available geo server1)
 (source-available country server3)
 (source-available seaport server4))
```

Table 2.5: The dynamic part of the initial state

4 Searching for High-Quality Plans

This information gathering task is naturally cast as a planning problem since the problem is to find a sequence of actions to retrieve and integrate the requested data. However, this task differs from much of the previous planning work in that the problem is to find a high-quality plan rather than any plan that solves the goal. In addition, since the cost of planning must be added to the overall processing time, the system must efficiently produce plans. This section describes how Sage efficiently generates high-quality plans.

The size of the search space is potentially quite large in this domain. Unlike many other planning domains, there are not a large number of interactions between operations. However, there are still a number of aspects to this problem that all contribute to the size of the search space. First, there may be a number of different possible orders in which the information can be processed. For example, if there are multiple sets of data that must be combined (i.e., a join), then there could be many possible orders in which to join the data. The order in which the data is combined is important since it determines the amount of intermediate data that will be produced. Second, there may be more than one source where the requested information can be retrieved and different sources may have different access costs. This may also affect what information must be processed locally and what processing can be done by the remote system. Third, there may be interactions between actions when two actions require the same resource. For example, if you have two queries to the same server, then these will have to be done sequentially. Fourth, there may be several alternative ways to combine information. For example, instead of retrieving two sets of data and performing a join, it may be more efficient to retrieve the first set of data, and then use the results to compose a query to the second source.

A natural approach to control the search is through the use of control rules. The control rules can be used to prune portions of the search space that are redundant or unnecessary to produce the best plan. In our initial implementation of the information gathering task we used the control rule language provided by UCPOP. Unfortunately, the overhead involved in considering the control rules was too high and greatly reduced the number of search nodes that could be considered in the same amount of time. This problem could probably be addressed by adding an efficient rule matching facility to UCPOP. We took the more direct approach of simply moving the control information directly into the definition of the functional predicates used in the operators. This type of control information prunes portions of the search space that will never need to be considered. For example, the operators consider only joins across data that are distributed in different information sources. It will generally be less efficient to pull two sets of information from the same information source and perform the join locally rather than in the remote source. In this case, this heuristic is captured in the implementation of the `join-partition` function. We carefully crafted the functional predicates to include this information so that they only generate choice points that are relevant to solving the problem.

In addition to constraining the space as much as possible, we also want to minimize the portion of the space that will need to be considered, but still consider alternative plans for processing the same query. To do this we use a branch-and-bound search with an evaluation function (specific to information gathering) for estimating the cost of alternative plans. Branch-and-bound expands the plan with the lowest estimated cost at each step in the search process, which will produce the optimal plan relative to the given operator definitions and evaluation function. The evaluation function employs standard database estimation techniques to estimate the cost of processing a query with a particular plan. The evaluation function estimates the cost of each operation by maintaining information about the size of each class and the number of different possible values for each role of a class (i.e., the cardinality of a role). Assuming a uniform distribution of the data, it then estimates the amount of intermediate data that will be retrieved and manipulated, which is usually the dominant cost in handling multidatabase queries. Using the estimated cost of each operation,

13
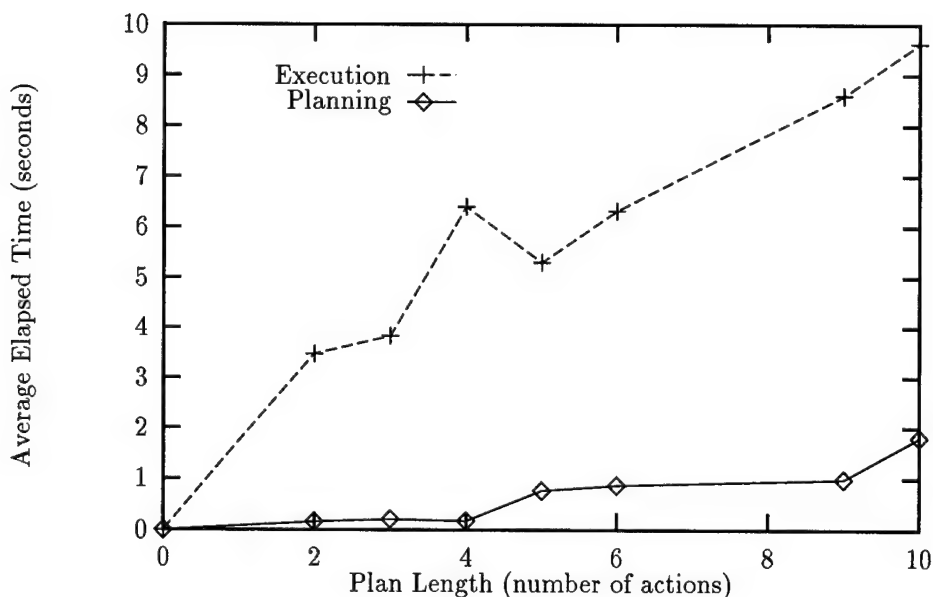
Figure 2.2: Comparison of the elapsed time for planning versus execution

the function computes an estimate for the overall cost of a plan, taking into account the parallelism of some of the actions. The evaluation function allows the planner to compare different partially-constructed plans; those plans that are more expensive than what the evaluation function determines to be the optimal plan will never be expanded further.

5   Results

The Sage planner serves as the query processing engine for the SIMS information mediator [7, 41]. Using SIMS we have built mediators to provide access to data for transportation planning and trauma-care medical information. We are also in the process of building mediators to integrate information for genetics counseling and military logistics. Note that for each of these applications the planner and planner domain remain the same, but the underlying information sources and the models of the information sources are changed.

In the transportation planning domain, the planner efficiently generates plans for queries that have as many as 30 terms and require access from up to three different information sources. To demonstrate that the system can efficiently generate plans for queries in this domain, Figure 2.2 compares the time to construct the plan for a query to the time for executing a plan (which does not include the planning time). The problems are ordered by plan size and range from 2 steps to 10 steps. The graph compares elapsed time and not CPU times since the execution of a plan occurs across multiple processes on different machines. Since there is some variation in elapsed time, each of the queries was run 10 times and the average time for each length plan is shown in the graph. The graph clearly shows that the planner can efficiently generate information gathering plans for the given test set. In addition, this time is only a small fraction of the overall execution time.

6   Related Work

The database community has been building specialized systems for performing query access planning for many years [34, 65]. In order to address this problem these systems exploit a variety of heuristics and techniques to constrain the search space. However, the problem is inherently a search problem, so there is no way to completely eliminate the search. These approaches have been carefully optimized for the problem of query processing in the single database environment and the distributed database environment. However,

14

there has been less work on the problem of query processing for multidatabase systems [43, 61] where there are a set of distributed, autonomous, and heterogeneous databases. The existing approaches in this environment are inflexible in the sense that choice of the information sources for a given query is fixed and the integration of this information is predefined. This essentially reduces the problem to the single-database query access planning problem.

The information gathering problem described here differs from the more traditional query access planning in two significant ways. First, instead of assuming that the choice of information source is fixed, as part of the planning process the system selects the information sources that will be used for processing a query. Second, traditional query access planning reasons about the query processing at a more detailed level which involves not just the selection of which operation to perform on the data, but also how to implement the operation efficiently (e.g., performing a join using a *sort join* or a *hash join*). Since we use other systems to actually implement the query processing actions, we leave it to these system to decide on the most efficient way to implement these operations. Instead, we have focused on the problem of deciding where to get the information, where to process it, and what order to do the processing.

A variety of work on planning has explored various aspects of the query access planning problem. The LADDER system [63] had a similar goal of integrating multiple sources of information. They developed a planner [27] based on a theorem prover that is used to find the minimal set of sources to cover a query, and they combined it with a specialized heuristic algorithm that produces an efficient query access plan. In contrast, Sage uses a general planner for the query access planning and integrates the source selection and query access planning into a single planning process, which allows the system to generate more efficient plans.

More recent work on query access planning has focused on plan merging and plan optimization. Qiang et al. [74] developed an approach to efficiently merging plans, including query access plans. Shekhar [66] developed an approach to trading off search time with execution time in optimizing query access plans. Both of these systems assume that the query access plans are given.

Another planner that has been built for a information gathering task is the XII planner [20]. This planner serves as the query processor for the Unix Softbot [22]. Compared to Sage, XII reasons about the information at a different level of granularity. Instead of representing general actions for manipulating data, each operator corresponds to a Unix command. The advantage of their approach is that it provides finer-grained control and reasoning of the information. The disadvantages are first, that the operators are application specific and a new set of operators would have to be engineered to support another application domain. Second, since the system reasons about the information as individual tuples instead of sets of information, it would be impractical to efficiently reason about and manipulate large amounts of data.

Levy et al. [48] present a different approach to information gathering. In their work they have been developing specialized algorithms for the problems of source selection and planning (essentially building a special-purpose planner). They have focused largely on efficient algorithms for source selection.

7 Discussion

This chapter described our experiences in applying a general-purpose planner to solve the information gathering problem. An important lesson from this work is that the issues that arise in this particular real-world problem are different than the ones many researchers have focused on in the past and continue to work on. Previous work on planning has focused extensively on both developing more expressive representations and handling interactions between operators. While these are certainly important problems, other issues such as constraining the space of plans so that it can be searched efficiently and generating high-quality plans have been largely ignored.

Consider the issue of constraining the space of plans. There are a number of speed-up learning systems that learn control knowledge to constrain the search space. But these systems do not fully address the problem since these systems often reduce a very large search to a smaller one, but in most cases they do not reduce search to the point where realistic problems in these domains can be solved. Other important work on using planning technology to solve real problems, such as in SIPE [73] and O-Plan[15], has relied on careful engineering of the domains such that problems can be solved with only a modest amount of search. When people build specialized planners for specific applications they develop techniques and heuristics that

make some interesting class of problems tractable. Planning research has focused heavily on the techniques and seems to have neglected the infrastructure required for developing and exploiting the heuristics required to solve real problems.

The issue of plan quality has received even less attention. Perhaps this is due to the fact that without good heuristics there is not much hope of doing anything but a satisficing search. However, in many real-world domains, (e.g., query access planning, process planning, logistics planning, etc.) finding a high-quality solution is an integral part of the problem.

The next step in our work will focus on gathering and integrating semi-structured information in the World Wide Web. In the Web we will have to deal with the problem of combining information from many more sources. In addition, sensing actions will probably play a larger role since a query may require gathering additional information in order to locate the relevant sources to answer a query [42]. We expect that the combination of larger plans and more operators will require us to push even harder on techniques for efficiently generating high-quality plans.

Chapter 3

Compiling Source Descriptions for Efficient and Flexible Information Integration

1 Introduction

The problem of integrating data from heterogeneous collections of sources is ubiquitous. With the rise of the Internet as well as intranets the number of available and relevant sources to an organization continues to grow. One effective solution to this problem is the development of information mediators [71]. An information mediator provides seamless access to a collection of related, but possibly heterogeneous and distributed data sources. There are a variety of approaches to building information mediators, illustrated by different approaches used in systems such as TSIMMIS [31], Garlic [30, 62], HERMES [1], Information Manifold [48], InfoSleuth [35], and SIMS [7, 9], to name a few. However, one issue that is common to all of these approaches is how to scale these systems to large numbers of information sources in a way that is both computationally tractable and natural to the developers of new applications.

In information mediators, a central problem is how to efficiently process queries. This query optimization problem consists of both selecting a set of sources that can be used to answer a query and generating a query plan that specifies the order of retrieval and manipulations on the data. In this chapter we focus on the first problem (also known as source selection) for a mediator with an expressive language for describing the contents of sources. In Chapter 2 we present our approach to generating query plans using a cost-based optimizer, which takes advantage of the source selection techniques presented here. Traditional cost-based optimization and complex operations, such as aggregations and sorting, can be naturally supported within our framework, although they are not addressed here. In this chapter we focus on the problem of how to compactly represent and efficiently use the alternative combinations of sources that can be used to answer queries posed to a mediator.

Our approach to source selection is to build a global domain model (sometimes referred to as a world model) for an application and describe the contents of each of the sources in terms of the domain model. Then the system automatically compiles the definitions of each of the sources into axioms that describe the possible ways the sources can be combined to produce any of the information that may be requested for each class in the domain model. The compilation algorithm is incremental, which means that when the available sources are changed, added, or deleted, the system can efficiently update the axioms. Our approach provides flexibility and source independence by describing sources in terms of the domain model and it provides an efficient mechanism to incorporate source selection into the query processing.

The approach presented in this chapter can be viewed as a combination of sources modeled as views on the domain model (e.g., Information Manifold [48]) and the domains classes modeled as views on the source models [43]. The advantage of the former approach (also known as view rewriting) is that each source is modeled independently of all the other sources, so new sources can be added and existing sources can be

17

modified without changing the domain model. The disadvantage is that performing the view rewritings is computationally hard and finding a complete answer to a query requires computing query containment in both directions. Moreover, this expensive computation is repeatedly done at query planning time. Our approach has the advantage of source independence without the computationally difficult problem of testing query containment during query planning. The advantage of the latter approach is that the sources required to provide the data for a specific class of information can be determined by simply looking up the definition of the domain class. The disadvantage is that determining these definitions is a difficult process to both build and maintain. Our approach provides the advantage of being able to quickly determine the combination of sources for a domain class since they are precompiled from the source definitions and avoids the disadvantage of having to manually build and maintain the definitions.

A specific instantiation of this general approach to source selection has been developed for the SIMS information mediator. This chapter describes the language used in SIMS for defining sources, the algorithms for compiling the domain axioms from the source definitions, the approach to instantiating these domain axioms at run-time, and the relationships with previous work. We also provide initial experimental results comparing this approach to source selection in SIMS with the previous approach that performed the source selection at run-time. Overall, our approach provides a simple, efficient, and elegant solution for integrating heterogeneous data sources.

2 Background

In the SIMS project [7, 9] we are addressing the problem of providing integrated access to heterogeneous distributed information sources. To build an application in SIMS, a user creates a *domain model* using the Loom [50] knowledge representation language and describes the source contents in terms of this model. The domain model establishes a fixed vocabulary describing object classes, their attributes, and the relationships among them. SIMS accepts queries in this high-level uniform language, processes these queries, and returns the requested data. Thus, the queries to SIMS do not contain information describing which sources are relevant to finding their answers or where they are located. Queries do not need to state how information obtained from different sources should be joined or otherwise combined or manipulated. It is the task of the system to determine how to efficiently and transparently retrieve and integrate the data necessary to answer a query.

In the previous work on SIMS [9] the selection of the sources was performed *dynamically* by searching the space of query reformulations given the domain model and source descriptions. This approach provided the flexibility we wanted in terms of dynamically selecting sources for answering queries; however, it did not scale well to large numbers of sources since the search space becomes quite large as the number of sources increases. The work described in the remainder of this chapter extends our previous work by precompiling the source definitions into a set of domain axioms. The domain axioms compactly express the possible ways of obtaining the data for each class in the domain model. This approach provides the same flexibility and extensibility of the previous approach and can perform the source selection much more efficiently.

In the remainder of this section, we review the language for representing a domain, describe how the sources are defined in terms of the domain model, and then present a detailed motivating example that is used throughout the remainder of this chapter.

2.1 Modeling the Domain and the Sources

An information mediator typically has a representation of its domain of expertise, called the *domain model*, and a set of *source models*, which are descriptions of information sources in terms of the domain model. In SIMS, we use a KL-ONE style knowledge representation language called Loom. KL-ONE style languages, also known as description logics, contain unary relations (*classes*), which represent the classes of the objects in the domain, and binary relations (*attributes*), which describe relationships between objects.

Classes can be defined as either primitive classes or they can be defined in terms of other classes. A primitive class is defined as a subclass without specifying the constraints that differentiate it from the parent class. For example, one might define large-seaports as a subclass of seaports without specifying what constraints differentiate it. In terms of modeling a set of sources, this is useful in the case where you have

two sources, where one is clearly a subclass of the other, but there is no simple way to characterize the specific subclass of information it contains.

A class can also be defined as a subclass of another class with the exact relationship specified as a set of constraints on the class. For example, large-seaport might be defined as a seaport with more than seven cranes. In addition, a class can be defined as the disjunction of a set of other classes. For example seaports might be broken down into two subclasses: large-seaports and small-seaports. Disjunction of classes (coverings) are useful for describing sets of sources that can be combined to produce a class of objects, such as seaports.

A set of attributes is associated with each class and any subclass of a given class, C, inherits all of the attributes of C. For purposes of integration, we also require that every class has at least one defined key, which represents one or more attributes that uniquely identify the objects in a class. Since there may be more than one way to uniquely identify a object, a class can also have multiple defined keys.

The domain model above is used to describe the available information sources. This is done by first constructing the corresponding domain class for a source and then specifying which attributes are provided by that source. A source description for source S with attributes $S.a_1^s, \ldots, S.a_n^s$ is written:

$$S(S.a_1^s, \ldots, S.a_n^s) = D_S(a_1^s, \ldots, a_n^s).$$

This specifies that the source S provides all instances of the class D_S with the corresponding attributes. In contrast to other approaches to information integration, such as in the Information Manifold [48], we assume that the source description defines exactly the class of information provided by the sources. This can be done without loss of generality because containment can be expressed by defining a subclass in the domain model. The advantage of exact descriptions are that it supports complete answers to queries, and when complete answers are not possible, it allows the system to determine when and in what way answers are incomplete. A limitation of our approach is that we cannot describe a source as a join over the domain classes, although a new domain class could be constructed and the source could be linked to the new class.

2.2 Motivating Example

Using the definitions of the previous section, we present a complete example of a domain and description of a set of sources. This example will be used throughout the remainder of the chapter to illustrate the basic ideas.

Consider a very simple application domain that contains a variety of information sources about various types of seaports. Figure 3.1 depicts the domain model constructed for this application domain. The domain classes are shown by ovals and are linked in an inheritance hierarchy; in the diagram, inheritance links are specified with solid arrows. As described above, a domain class can be primitively defined in terms of another class, or it can be defined precisely in terms of another class by specifying a set of constraint expressions. For example, the figure shows that a large seaport is defined as a subclass of seaport where the number of cranes available is greater than seven. A class can also be defined to be equivalent to a covering (union) of two or more other domain classes. For example, seaport is defined as the union of large and small seaports. Every class may have a corresponding set of attributes, shown by arrows, and classes also inherit all of the attributes of the classes above it. In this case seaport has four attributes, geographic location code (gc), port name (pn), country name (cn), and number of cranes (cr). The class of European large seaports inherits all of these attributes and also has an additional attribute, European code (ec), which is specific to this class.

In addition to the domain model, the available sources are also shown in the figure. These are shown by the database symbols linked to the corresponding domain classes by dashed lines. The link between the source and the domain class means that there is a one-to-one mapping between the instances of the domain class and the instances in the source. In the example, there are three sources for the class large seaport: the first source, s2, provides the attribute pn, the second source, s3, provides the source gc, and the third source, s7, provides the attributes pn and cn. For the sake of simplicity, each information source described here will be assumed to have only a single table in it; both the source and the table will be referred to by the same name. Since sources might provide only a subset of the possible attributes of a class, the specific attributes for each source are also shown in the figure.

There are a total of seven data sources, which provide different sets of attributes about components of this domain, none of which provide all attributes for all classes. In the next section, we will explain how the

19
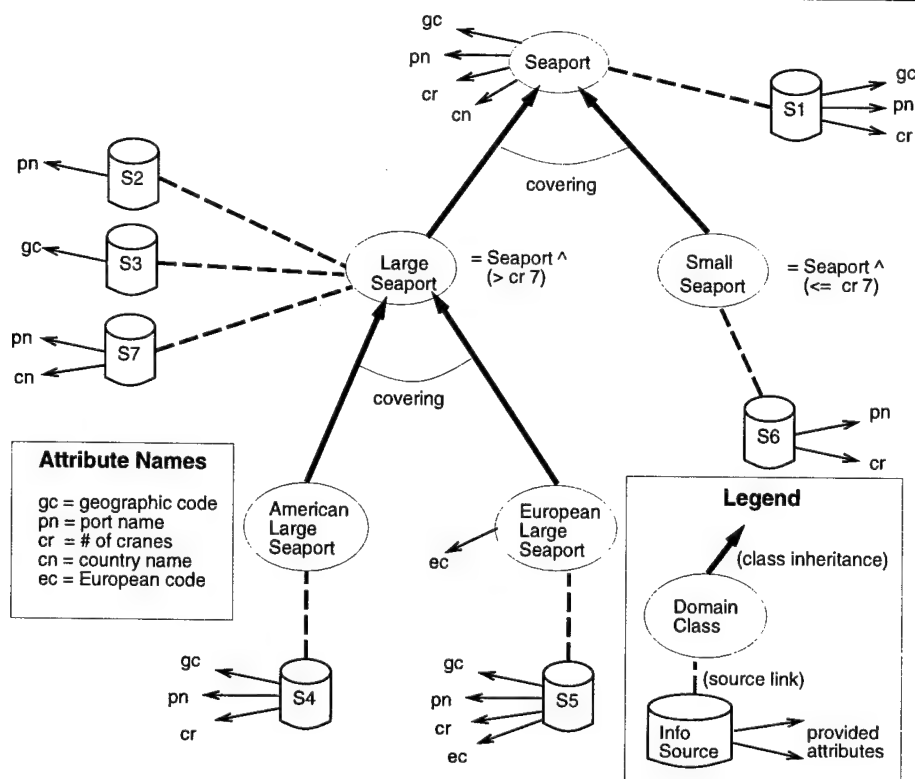
Figure 3.1: Example Domain Model

sources definitions shown in the example are compiled into axioms that specify how to combine the sources
to produce the data for domain classes.

3   Compiling Domain Axioms from Source Definitions

A fundamental task of a mediator is to translate a query expressed in terms of the domain model into
queries to the underlying sources. This involves finding the relevant combinations of sources that provide
the attributes required by each class in the query. Instead of repeatedly searching for these combinations at
run-time for each query, our system compiles in advance a set of *domain axioms* that compactly capture these
combinations. Using these axioms, the system can improve the efficiency of query planning considerably
because the optimizer can use the readily available axioms as macro expansions avoiding the search involved
in constructing the combinations of sources for the domain classes. Moreover, the compilation effort is
amortized over all subsequent queries on the domain model and sources. This section describes the details
of how the domain axioms are pre-compiled and stored for efficient use at run-time.

A domain axiom specifies a particular way in which the available sources can be combined to provide
the data for a domain class. For example, the port-name for the class large-seaport can be directly retrieved
from the source s2, as can be seen in Figure 3.1. This would be expressed as the axiom:

large-seaport(pn) $\equiv$ s2(s2.pn)

There may be several axioms for a given class and set of attributes. For example, an alternative way of
obtaining the previous data for large-seaport is to perform a union over sources s4 and s5, which provide a
covering for the class, resulting in the axiom:

large-seaport(pn) $\equiv$ s4(s4.pn) $\vee$ s5(s5.pn)

Since the number of possible combinations of attributes can be very large even for a single class, the
system does not compute all possible axioms for all combinations of attributes. Instead, the axiom generation
proceeds in a two phases. In the first phase, the system compiles, off-line, the minimal set of domain axioms.

We call this set minimal because all valid domain axioms for each class and for each set of attributes can be derived from it. Each axiom in this set provides as much information, as many attributes, as possible for its particular combination of sources. For example, the previous axiom does not belong to the minimal set, because two more attributes, cr and gc, can be obtained from the combination of s4 and s5. The following axiom, which incorporates all the attributes for this particular combination of sources, will be included in the minimal set:

large-seaport(cr gc pn) ≡ s4(s4.cr s4.gc s4.pn) ∨ s5(s5.cr s5.gc s5.pn)

In the second phase, the system organizes the axioms for each domain class into a lattice. Each lattice is a partial order, using set containment, over subsets of attributes for a domain class. This data structure caches the axioms that have already been computed and makes it possible to efficiently derive, at runtime, new axioms for a given class and set of attributes in response to user queries. For example, if a query requests large-seaport(gc pn), the corresponding axiom(s) will be derived from the stored axiom(s) for large-seaport(cr gc pn). Section 3.1 describes the compilation of the minimal domain axioms and Section 3.2 describes how the lattice structure is constructed and subsequently used during query processing.

3.1 Automatically Compiling the Minimal Set of Domain Axioms

The system automatically compiles the minimal set of domain axioms by applying a set of five inference rules. Each rule captures an orthogonal type of inference about how sources can be combined based on our representation language. The rules use the available source descriptions and the domain model. The five rules are:

- **Direct**: Translates the supplied source definitions into axioms;
- **Covering**: Exploits the covering relationships in the domain model;
- **Definition**: Exploits the constraints in the definition of a domain class ;
- **Inherit**: Exploits the inheritance of superclass attributes via shared keys; and
- **Compose**: Combines axioms on a given class to provide additional attributes.

Our compilation algorithm first applies the Direct rule, and then the remaining four rules apply in parallel until quiescence. Our algorithm is incremental, similar in spirit to the semi-naive evaluation of logic programs. In the application of each rule at least one of the axioms involved must belong to the most recent generation. Also, in order to avoid unnecessary computation, redundant and subsumed axioms are eliminated at each generation. To facilitate this process the axioms are stored in a normal form (see below). The remainder of this section will explain each of these rules in turn on the example presented in Section 2.2. For purposes of exposition we will describe the application of the inference rules in a particular order.

The Direct Rule The Direct rule installs each source declaration, converted into the axiom representation, into the axioms list for each class. An example installed axiom is:

seaport(cr gc pn) ≡ s1(s1.cr s1.gc s1.pn) ∧ cr = s1.cr ∧ gc = s1.gc ∧ pn = s1.pn   *(1.1)*

This axiom specifies that one way to obtain the number of cranes (cr), geoloc-code (gc), and port-name (pn) of seaport is using the information source s1. Note that the axiom is expressed in terms of equivalence (≡), not containment; that is, we are declaring that s1(s1.cr s1.gc s1.pn) is co-extensional with seaport(cr gc pn), but as explained before, this does not limit the generality of our representation because containment can be expressed by subclassing in the domain model. The series of equality constraints on the right-hand side detail the exact mapping between the domain-level attributes {cr, gc, pn} and the source attributes {s1.cr s1.gc s1.pn}. Hereafter in the presentation, we will elide these binding equality constraints.[1]

After the execution of the Direct rule, seven domain axioms have been associated with the classes of our sample domain, as shown in Figure 3.2. They correspond precisely to the source definitions depicted in Figure 3.1.

---

[1] The naming convention used in the chapter, where an attribute A of some concept always corresponds to a source attribute named S.A in source S is not required by the system, but may assist the reader in understanding the presented axioms.

seaport(cr gc pn)	≡	s1(s1.cr s1.gc s1.pn)	1.1
small-seaport(cr pn)	≡	s6(s6.cr s6.pn)	1.2
large-seaport(gc)	≡	s3(s3.gc)	1.3
large-seaport(pn)	≡	s2(s2.pn)	1.4
large-seaport(cn pn)	≡	s7(s7.cn s7.pn)	1.5
american-large-seaport(cr gc pn)	≡	s4(s4.cr s4.gc s4.pn)	1.6
european-large-seaport(cr ec gc pn)	≡	s5(s5.cr s5.ec s5.gc s5.pn)	1.7

Figure 3.2: Axiom state after application of the Direct rule.

The Covering Rule When a class in the domain model is defined as being equivalent to a covering of two or more of its subclasses, we can use this definition to generate new axioms for the parent class based on the axioms of its subclasses. The rule retrieves the axioms for each subclass of the given covering and forms the Cartesian product of these axioms sets. For each tuple of axioms in the Cartesian product, it computes the intersection of the attributes provided by each axiom. If the intersection is not empty, a new axiom providing the common attributes is generated. Then, from each axiom in the tuple, we project out any attributes not included in the intersection. The body of the new axiom is the disjunction of the projected axioms. For example, consider the following covering in the domain model:

large-seaport ≡ american-large-seaport ∨ european-large-seaport.

The Covering rule retrieves axiom *(1.6)* for american-large-seaport which provides cr, gc, and pn, and axiom *(1.7)* for european-large-seaport which provides cr, ec, gc, and pn. The intersection set is {cr, gc, pn}. Both axioms are projected through this intersection (which in this case simply amounts to removing the attribute ec from the axiom for european-large-seaport) and combined by disjunction. The final axiom is:

large-seaport(cr gc pn) ≡ s4(s4.cr s4.gc s4.pn) ∨ s5(s5.cr s5.gc s5.pn) *(2.4)*

The Covering rule is applied across the class hierarchy in a bottom-up fashion. This order ensures that a covering axiom produced at a lower level could participate in a later covering at a higher level. For example, a covering axiom for large-seaport, computed from american-large-seaport and european-large-seaport, could form part of a second covering for seaport. The specification of the algorithm used to process the coverings is presented in Figure 3.3. In our example domain, after processing the coverings throughout the entire hierarchy four more axioms are added, for a total of 11, as shown in Figure 3.4.

$\forall C = C_1 \lor C_2 \lor \cdots \lor C_i \lor \cdots \lor C_n = \bigvee_i C_i, C \in \mathbf{C}, C_i \in \mathbf{C}$, (*C* from the hierarchy leaves to the root)

    For each element $\{...[a_{1k}(\bar{x}_{1k}), a_{2l}(\bar{x}_{2l}), ..., a_{nm}(\bar{x}_{nm})]...\} \in A_1 \times \cdots \times A_n, A_i = \text{AXIOMS}(C_i)$

        Let $\bar{x} = (\bar{x}_{1k} \cap \bar{x}_{2l} \cap \cdots \cap \bar{x}_{nm})$.

           If $\bar{x} \neq \emptyset$ then add axiom $a(\bar{x}) = \text{PROJECTION}(a(\bar{x}_{1k}), \bar{x}) \lor$
                                            $\text{PROJECTION}(a(\bar{x}_{2k}), \bar{x}) \lor$
                                            $\cdots \lor$
                                            $\text{PROJECTION}(a(\bar{x}_{nm}), \bar{x})$

        to AXIOMS(*C*).

Notation:

    C is the set of all domain classes in the model hierarchy.
    C is a particular domain class, $C \in \mathbf{C}$.
    AXIOMS(*C*) holds the set of axioms for class *C*; it can be updated.
    $a(\bar{x}) \in$ AXIOMS(*C*) is a particular axiom, which provides attribute set $\bar{x}$.
    PROJECTION$(a(\bar{x}), \bar{y}), \bar{y} \subseteq \bar{x}$ is a subroutine which eliminates from $a(\bar{x})$ any attributes or terms not needed to
        obtain all of $\bar{y}$.

Figure 3.3: Algorithm for the Covering rule

The Definition Rule This rule exploits the constraints in the definitions of a domain class. Essentially, when a class is defined in terms of a parent class and a set of constraints, the rule generates axioms for the class by conjoining the axioms of the parent class with the constraints in the class definition. Note that the axiom of the parent class needs to provide all attributes used in the definition constraints. Also, to avoid

seaport(pn)	≡	s2(s2.pn) ∨ s6(s6.pn)	*2.1*
	≡	s6(s6.pn) ∨ s7(s7.pn)	*2.2*
seaport(cr pn)	≡	s4(s4.cr s4.pn) ∨ s5(s5.cr s5.pn) ∨ s6(s6.cr s6.pn)	*2.3*
seaport(cr gc pn)	≡	s1(s1.cr s1.gc s1.pn)	*1.1*
small-seaport(cr pn)	≡	s6(s6.cr s6.pn)	*1.2*
large-seaport(gc)	≡	s3(s3.gc)	*1.3*
large-seaport(pn)	≡	s2(s2.pn)	*1.4*
large-seaport(cn pn)	≡	s7(s7.cn s7.pn)	*1.5*
large-seaport(cr gc pn)	≡	s4(s4.cr s4.gc s4.pn) ∨ s5(s5.cr s5.gc s5.pn)	*2.4*
american-large-seaport(cr gc pn)	≡	s4(s4.cr s4.gc s4.pn)	*1.6*
european-large-seaport(cr ec gc pn)	≡	s5(s5.cr s5.ec s5.gc s5.pn)	*1.7*

Figure 3.4: Axiom state after application of the Covering rule (new axioms in bold).

generating non-minimal axioms, the rule does not consider any parent axiom which includes a source for the current class.[2] This occurs when the parent axiom was generated by a covering that includes the child class. The attributes of the parent axiom can only be a subset of those provided directly by sources to the current class, and those axioms are already installed by the Direct rule. For example, consider the following definition from the domain model:

small-seaport $\equiv$ seaport $\wedge$ cr $\leq$ 7.

The Definition rule considers axiom *(1.1)* for seaport which provides the attribute cr needed for the constraint cr $\leq$ 7. Since *(1.1)* did not result from a covering over small-seaport, the rules yields the following axiom:

small-seaport(cr gc pn) $\equiv$ s1(s1.cr s1.gc s1.pn) $\wedge$ s1.cr $\leq$ 7 *(3.1)*

Note that the other three axioms for seaport are built from coverings involving the source s6 for small-seaport, so they would not provide more attributes than what s6 provides directly, and the definition rule does not apply.

The Definition rule is applied top-down across each class hierarchy of the domain model. Processing in this order means that a given class may exploit definition axioms created in terms of axioms from its ancestors as well as its direct parent, merely by processing the definition rule over any relevant axioms stored with the parent. The specification of the algorithm used to handle class definitions is shown in Figure 3.5. In our example domain, the application of the Definition rule generates two new axioms, for a total of 13, as shown in Figure 3.6.

The Inherit Rule When a class in the domain model shares a key with one of its ancestor classes which has sources for some attribute not available in the sources of a descendant class, the Inherit rule joins axioms from each class over the shared key in order to provide the new attributes to the descendant. Intuitively, the information about an ancestor class can be transferred to the descendant subclass as long as the system has a way of identifying those objects that belong only to the subclass. In order to do so, this rule conjoins an ancestor axiom with a source for the subclass, similarly to the previous rule, Definition, that conjoined an ancestor axiom with the definition constraints of the subclass. For example, there is no axiom for class american-large-seaport that provides the attribute cn, but axiom *(1.5)* for large-seaport does provide it, and moreover, axioms for american-large-seaport and large-seaport share a key (e.g., {pn}). Therefore, this rule generates the following axiom which brings cn down to american-large-seaport:

american-large-seaport(cn cr gc pn) $\equiv$ s4(s4.cr s4.gc s4.pn) $\wedge$ s7(s7.cn s7.pn) *(4.5)*

The Inherit rule applies bottom-up across the class hierarchy and looks at *all* ancestors of each class. In this way, it considers all possible class/superclass combinations without ever encountering any axioms generated by previous applications of the same rule to intervening classes. The algorithm for the Inherit rule is shown in Figure 3.7. For our example domain, this rule generates six additional axioms, for a total of 19, as shown in Figure 3.8.

The Compose Rule The Compose rule ensures that the axioms for a domain class contain as many attributes as possible. For each pair of axioms for a given class which share a key[3] and each provides attributes that the other does not, we can create an axiom which pools their two sets of attributes. For example, axioms *(1.5)* and *(2.4)* for large-seaport should be composed

---

[2] In our system this type of reasoning is efficient. Because axioms are generated from particular rules, it is a simple matter to record the proof tree for each generated axiom. This proof tree can be examined to answer such questions. The details of this proof tree representation have been omitted from the discussion for brevity.

[3] Actually, for which a *path* of keys can be constructed.

$\forall C = C' \wedge c_1 \wedge c_2 \wedge \cdots \wedge c_n, C \in \mathbf{C}, C' \in \mathbf{C},$
  where C' is an (immediate) parent class of C and
  each constraint $c_i = (b_i \ \theta \ m_i)$
    with $b_i \in \text{ATTRIBUTES}(C_i)$,
    $\theta \in \{>, \geq, <, \leq, =, \neq\}$,
    and constant m_i
  Let $\bar{x} = \{b_i : 1 \leq i \leq n\}$ be the set of all domain attributes used
  in all definitional constraints for C
  $\forall a_k'(\bar{x}_k) \in \text{AXIOMS}(C')$
    if $\bar{x} \subseteq \bar{x}_k$ and a_k' was not derived from a covering which includes C,
    then add axiom $a(\bar{x}_k \cup \bar{x}) = a_k'(\bar{x}_k) \wedge c_1 \wedge c_2 \wedge \cdots c_n$ to $\text{AXIOMS}(C)$

Notation:

  C denotes a domain class.
  $\text{ATTRIBUTES}(C)$ is the set of all attributes defined in a particular domain class C.
  $b \in \text{ATTRIBUTES}(C)$ or $b \in \bar{x}$ is a particular domain attribute.
  c denotes a constraint expression used in a class definition. Order constraints (e.g., $(CR > 7)$), equality constraints
    (e.g., (PN = "Long Beach")) and inequality constraints (e.g., (GC $\neq$ "XJJD")) between a single attribute and
    constant are currently supported.
  θ is an operator used in a constraint.
  m is a constant used in a constraint.

Figure 3.5: Algorithm for the Definition rule

seaport(pn)	$\equiv$	s2(s2.pn) $\vee$ s6(s6.pn)	*2.1*
	$\equiv$	s6(s6.pn) $\vee$ s7(s7.pn)	*2.2*
seaport(cr pn)	$\equiv$	s4(s4.cr s4.pn) $\vee$ s5(s5.cr s5.pn) $\vee$ s6(s6.cr s6.pn)	*2.3*
seaport(cr gc pn)	$\equiv$	s1(s1.cr s1.gc s1.pn)	*1.1*
small-seaport(cr pn)	$\equiv$	s6(s6.cr s6.pn)	*1.2*
small-seaport(cr gc pn)	$\equiv$	**s1(s1.cr s1.gc s1.pn) $\wedge$ s1.cr $\leq$ 7**	*3.1*
large-seaport(gc)	$\equiv$	s3(s3.gc)	*1.3*
large-seaport(pn)	$\equiv$	s2(s2.pn)	*1.4*
large-seaport(cn pn)	$\equiv$	s7(s7.cn s7.pn)	*1.5*
large-seaport(cr gc pn)	$\equiv$	s4(s4.cr s4.gc s4.pn) $\vee$ s5(s5.cr s5.gc s5.pn)	*2.4*
	$\equiv$	**s1(s1.cr s1.gc s1.pn) $\wedge$ s1.cr $>$ 7**	*3.2*
american-large-seaport(cr gc pn)	$\equiv$	s4(s4.cr s4.gc s4.pn)	*1.6*
european-large-seaport(cr ec gc pn)	$\equiv$	s5(s5.cr s5.ec s5.gc s5.pn)	*1.7*

Figure 3.6: Axiom state after application of the Definition rule (new axioms in bold)

together because they share the key {pn}, and attributes {cn} and {gc, cr} appear uniquely in *(1.5)* and *(2.4)* respectively. The conjunction of the two axiom formulas generates the following axiom (shown in disjunctive normal form):

  large-seaport(cn cr gc pn) $\equiv$ [s4(s4.cr s4.gc s4.pn) $\wedge$ s7(s7.cn s7.pn) $\wedge$ s4.pn = s7.pn] $\vee$
            [s5(s5.cr s5.gc s5.pn) $\wedge$ s7(s7.cn s7.pn) $\wedge$ s5.pn = s7.pn]   *(5.1)*

  The specification of the algorithm for Compose rule is presented in Figure 3.9. Because this rule processes only a single class at a time, it can be applied over the class hierarchy in an arbitrary order. Figure 3.10 shows the axioms after the Compose rule is applied. These are all the axioms for our example domain. The compilation process generates a total twenty axioms from the seven initial source descriptions for the five domain classes.

$\forall C \in \mathbf{C}$ having ancestor $C' \in \mathbf{C}, \text{KEYS}(C) \cap \text{KEYS}(C') \neq \emptyset,$
  $\forall a_i(\bar{x}_i) \in \text{AXIOMS}(C)$
    $\forall a_j'(\bar{x}_j') \in \text{AXIOMS}(C')$
      where $\bar{x}_j' - \bar{x}_i \neq \emptyset$
    If $\exists \bar{k} \subseteq \bar{x}_j' \cap \bar{x}_i \wedge \bar{k} \in \text{KEYS}(C) \wedge a_j'(\bar{x}_j')$ was not derived from a covering including C
      then add axiom $a_{\bar{k}}(\bar{x}_i \cup \bar{x}_j') = a_i \wedge a_j$ to $\text{AXIOMS}(C)$.

Figure 3.7: Algorithm for the Inherit rule

seaport(pn)	≡	s2(s2.pn) ∨ s6(s6.pn)	2.1
	≡	s6(s6.pn) ∨ s7(s7.pn)	2.2
seaport(cr pn)	≡	s4(s4.cr s4.pn) ∨ s5(s5.cr s5.pn) ∨ s6(s6.cr s6.pn)	2.3
seaport(cr gc pn)	≡	s1(s1.cr s1.gc s1.pn)	1.1
small-seaport(cr pn)	≡	s6(s6.cr s6.pn)	1.2
small-seaport(cr gc pn)	≡	s1(s1.cr s1.gc s1.pn) ∧ s1.cr ≤ 7	3.1
	≡	**s1(s1.cr s1.gc s1.pn) ∧ s6(s6.cr s6.pn) ∧ s1.pn = s6.pn**	4.1
large-seaport(gc)	≡	s3(s3.gc)	1.3
large-seaport(pn)	≡	s2(s2.pn)	1.4
large-seaport(cn pn)	≡	s7(s7.cn s7.pn)	1.5
large-seaport(cr gc pn)	≡	s4(s4.cr s4.gc s4.pn) ∨ s5(s5.cr s5.gc s5.pn)	2.4
	≡	s1(s1.cr s1.gc s1.pn) ∧ s1.cr > 7	3.2
	≡	**s1(s1.cr s1.gc s1.pn) ∧ s3(s3.gc) ∧ s1.gc = s3.gc**	4.2
	≡	**s1(s1.cr s1.gc s1.pn) ∧ s2(s2.pn) ∧ s1.pn = s2.pn**	4.3
large-seaport(cn cr gc pn)	≡	**s1(s1.cr s1.gc s1.pn) ∧ s7(s7.cn s7.pn) ∧ s1.pn = s7.pn**	4.4
american-large-seaport(cr gc pn)	≡	s4(s4.cr s4.gc s4.pn)	1.6
american-large-seaport(cn cr gc pn)	≡	**s4(s4.cr s4.gc s4.pn) ∧ s7(s7.cn s7.pn) ∧ s4.pn = s7.pn**	4.5
european-large-seaport(cr ec gc pn)	≡	s5(s5.cr s5.ec s5.gc s5.pn)	1.7
european-large-seaport(cn cr ec gc pn)	≡	**s5(s5.cr s5.ec s5.gc s5.pn) ∧ s7(s7.cn s7.pn) ∧ s5.pn = s7.pn**	4.6

Figure 3.8: Axiom state after application of the Inherit rule (new axioms in bold).

$\forall C \in \mathbf{C}$,
    Divide AXIOMS (C)
        into a set $E = \{...\bar{x}_k : \{a_{k,1}(\bar{x}_k), a_{k,2}(\bar{x}_k), ...\},$
                 $\bar{x}_{k+1} : \{a_{k+1,1}(\bar{x}_{k+1}), a_{k+1,2}(\bar{x}_{k+1}), ...\}...\}$
        of equivalence classes over the attribute sets, ordered first by increasing size and then lexicographically.
    $\forall$ (pairs of classes) e_k, e_l drawn from $E, l > k$
        $\forall a_i \in e_k$
            $\forall a_j \in e_l$
                $\forall \bar{k} \in$ KEYS (C)
                    If $\bar{k} \subseteq \bar{x}_k \cap \bar{x}_l$
                    Then create axiom $a'(\bar{x}_k \cup \bar{x}_l) = a_i \wedge a_j$
                    Insert a' into the equivalence class for $\bar{x}_k \cup \bar{x}_l$, maintaining lexicographic order.
    Finally, transfer all axioms in E to AXIOMS (C).

Figure 3.9: Algorithm for the Compose rule

Normalization, Simplification and Projection Applying the rules above requires determining whether two computed axioms are equivalent. For example, the Compose rule might independently reconstruct an axiom that had already been computed by previous rules. In order to remove redundant axioms it is necessary to put the axioms into a normal form. We have chosen a sorted disjunctive normal form as our basic representation. Once the formula is in this normal form, we remove replicated predicates and implied order predicates in the conjunctions and across disjunctions.

An axiom provides a set of attributes for a class C. Obviously, it also provides all subsets of these attributes. It is useful to find the simplified expression of an axiom $a(\bar{x})$ when only a subset $\bar{y}$ of its attributes are required. We call the resulting axiom $a'(\bar{y})$ the *projection* of $a(\bar{x})$ on $\bar{y}$. First, we present some examples of projection intuitively. Then, we give a formal definition. Consider the axiom:

   large-seaport(cn cr gc pn) ≡ [s4(s4.cr s4.gc s4.pn) ∧ s7(s7.cn s7.pn) ∧ s4.pn = s7.pn] ∨
                             [s5(s5.cr s5.gc s5.pn) ∧ s7(s7.cn s7.pn) ∧ s5.pn = s7.pn]   *(5.1)*

The result of the projection on {cr, gc, pn} is:

   large-seaport(cr gc pn) ≡ s4(s4.cr s4.gc s4.pn) ∨ s5(s5.cr s5.gc s5.pn)   *(5.1′)*

Informally, the reasoning behind this projection is as follows. Given our source descriptions, either of s7(s7.cn s7.pn) or [s4(s4.cr s4.gc s4.pn) ∨ s5(s5.cr s5.gc s5.pn)] is enough to ensure that the data retrieved by *(5.1′)* are in the large-seaport class. But we only need the attributes {cr, gc, pn} which can all be obtained from the disjunction of s4 and s5. Therefore, s7 is unnecessary and the projected axiom can be simplified. This example shows that a predicate appears in an axiom essentially because either it forms part of the formula that proves that the axiom is equivalent to the given concept or it contributes some of the needed attributes. Consider the axiom *(3.2)*:

   large-seaport(cr gc pn) ≡ s1(s1.cr s1.gc s1.pn) ∧ s1.cr > 7   *(3.2)*

The projected axiom on {pn} is:

   large-seaport(pn) ≡ s1(s1.cr s1.pn) ∧ s1.cr > 7   *(3.2′)*

seaport(pn)	≡ s2(s2.pn) ∨ s6(s6.pn)	*2.1*
	≡ s6(s6.pn) ∨ s7(s7.pn)	*2.2*
seaport(cr pn)	≡ s4(s4.cr s4.pn) ∨ s5(s5.cr s5.pn) ∨ s6(s6.cr s6.pn)	*2.3*
seaport(cr gc pn)	≡ s1(s1.cr s1.gc s1.pn)	*1.1*
small-seaport(cr pn)	≡ s6(s6.cr s6.pn)	*1.2*
small-seaport(cr gc pn)	≡ s1(s1.cr s1.gc s1.pn) ∧ s1.cr ≤ 7	*3.1*
	≡ s1(s1.cr s1.gc s1.pn) ∧ s6(s6.cr s6.pn) ∧ s1.pn = s6.pn	*4.1*
large-seaport(gc)	≡ s3(s3.gc)	*1.3*
large-seaport(pn)	≡ s2(s2.pn)	*1.4*
large-seaport(cn pn)	≡ s7(s7.cn s7.pn)	*1.5*
large-seaport(cr gc pn)	≡ s4(s4.cr s4.gc s4.pn) ∨ s5(s5.cr s5.gc s5.pn)	*2.4*
	≡ s1(s1.cr s1.gc s1.pn) ∧ s1.cr > 7	*3.2*
	≡ s1(s1.cr s1.gc s1.pn) ∧ s3(s3.gc) ∧ s1.gc = s3.gc	*4.2*
	≡ s1(s1.cr s1.gc s1.pn) ∧ s2(s2.pn) ∧ s1.pn = s2.pn	*4.3*
large-seaport(cn cr gc pn)	≡ s1(s1.cr s1.gc s1.pn) ∧ s7(s7.cn s7.pn) ∧ s1.pn = s7.pn	*4.4*
	≡ **[s4(s4.cr s4.gc s4.pn) ∧ s7(s7.cn s7.pn) ∧ s4.pn = s7.pn] ∨**	
	[s5(s5.cr s5.gc s5.pn) ∧ s7(s7.cn s7.pn) ∧ s5.pn = s7.pn]	*5.1*
american-large-seaport(cr gc pn)	≡ s4(s4.cr s4.gc s4.pn)	*1.6*
american-large-seaport(cn cr gc pn)	≡ s4(s4.cr s4.gc s4.pn) ∧ s7(s7.cn s7.pn) ∧ s4.pn = s7.pn	*4.5*
european-large-seaport(cr ec gc pn)	≡ s5(s5.cr s5.ec s5.gc s5.pn)	*1.7*
european-large-seaport(cn cr ec gc pn)	≡ s5(s5.cr s5.ec s5.gc s5.pn) ∧ s7(s7.cn s7.pn) ∧ s5.pn = s7.pn	*4.6*

Figure 3.10: Axiom state after application of Compose rule (new axioms in bold). Minimal set of domain axioms.

The constraint s1.cr > 7 is needed to ensure that the seaports provided by s1 are indeed large-seaports. In order to test this constraint the attribute s1.cr has to be retrieved from s1 even though it is not one of the requested attributes. Other examples of projection appear in Section 3.2.

Formally, a subformula g of an axiom $a = g \wedge r$ for C ($a \equiv C$) is called a *core* of a if $g \equiv C$ and no subformula of g is also a core. In other words, g is a minimal subformula of a that entails equivalence to the class C. Our system efficiently computes the cores of an axiom by using the derivation proof (based on the compilation rules) of the axiom. Recursively, the cores of a compose are the cores of its components, the cores of a covering are the disjunction of the cores of components, the core of a definition is the set of constraints, and the core of a direct is the direct source. For example, axiom *(5.1)* is obtained by applying the Compose rule to a Covering axiom and a Direct axiom. Therefore, the two cores are s7(s7.cn s7.pn), direct, and [s4(s4.cr s4.gc s4.pn) ∨ s5(s5.cr s5.gc s5.pn)], disjunction of direct cores for each of the subclasses in the covering.

The *projection* of an axiom $a(\bar{x})$ for a class C on the set of attributes $\bar{y}$, $\bar{y} \subseteq \bar{x}$, is an axiom $a'(\bar{y})$ that satisfies:

1. All predicates of a' appear in a
2. a' contains at least one core of a
3. a' contains more than one core only when each core uniquely provides some attribute in $\bar{y}$
4. All predicates in the conjunctions of a' are connected. Note that connections only occur either between ordinary predicates that share a common key, or between an ordinary predicate and an order predicate on the attribute used in the order predicate.
5. The only attributes in the body of a' are those needed to satisfy the previous condition or are requested in $\bar{y}$.

3.2 Using the Domain Axioms Efficiently

Each axiom in the minimal axiom set provides as many attributes as possible for its particular combination of sources. However, a user query may request any subset of the attributes of a domain class. Therefore new axioms for the desired attributes have to be efficiently derived from the minimal set – or signal that the query is unsatisfiable because there are no sources for those attributes. To that effect, the axioms for each class in the domain model are organized in a lattice. Each node in the lattice holds the axioms that represent all alternative ways of obtaining a particular set of attributes. The edges of the lattice capture set containment on attributes. An example of an axiom lattice that contains only the minimal axioms for the class large-seaport appears in Figure 3.11.

A lattice for a domain class is constructed in two phases. In the first phase, the minimal set of axioms is transferred to the lattice and the nodes are completed with *supplementary* axioms. The supplementary axioms are needed to ensure that each node holds all possible axioms for its attribute set. This phase is performed still at axiom compilation time. In the second phase, the lattice is used as an axiom cache and new axioms are generated as demanded by user queries. If a node for the requested attributes for a class is already present in the lattice, the axioms of the node are returned. Otherwise, a new node and corresponding axioms are computed. We refer to these as *interstitial* axioms since they lie between previous axiom sets (logically and in the lattice).

Supplementary Axioms The minimal axioms in a node may not list all the possible ways of combining the available sources to provide the attributes of the node. For example, in Figure 3.11, there is one additional way to obtain large-seaport(cr,
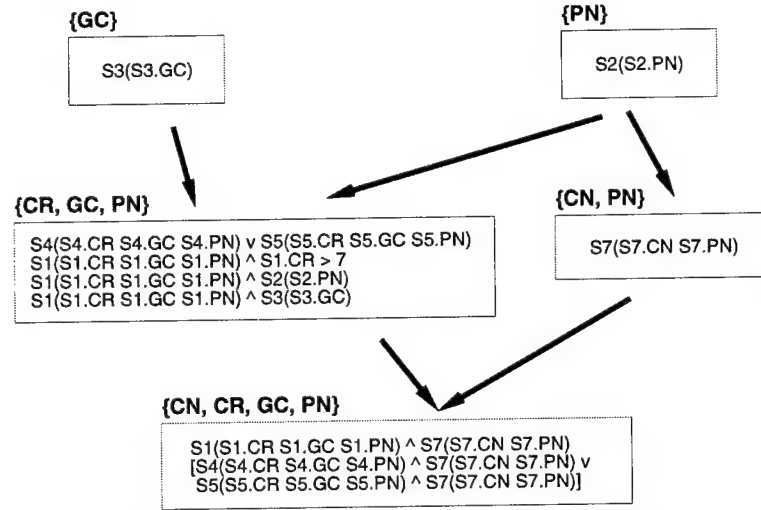
26

Figure 3.11: Initial axiom lattice for large-seaport (with minimal axioms only).

gc, pn) by combining sources s1 and s7. However the minimal axiom that combines s1 and s7, *(4.4)*, provides more attributes, {cn, cr, gc, pn}, and it is found in another node of the lattice. Fortunately, the desired axiom is easily computed from *(4.4)* by projection:

large-seaport(cr, gc, pn) $\equiv$ s1(s1.cr s1.gc s1.pn) $\wedge$ s7(s7.pn)

Supplementary axioms are computed when the lattice is first constructed. For a given class the axioms are introduced in the lattice in order of decreasing number of attributes. This corresponds to filling the diagram in Figure 3.11 bottom up. As each node is added to the lattice, the axioms in the superset (child) nodes are examined. Any projection of a superset axiom into the attributes of the current node, that is not equivalent to any of the axioms already present, is added to the node. The specification of the algorithm for computing the supplementary axioms is shown in Figure 3.12. The supplementary axioms for the example lattice are marked with an asterisk (*) in Figure 3.13.

$\forall C \in \mathbf{C}$
        Create a lattice $\mathcal{L}$ for C.
        Sort AXIOMS (C) by decreasing number of attributes, then lexicographically
        $\forall$ axioms $a(\bar{x}) \in$ SORTED AXIOMS (C)
                If a node N providing $\bar{x}$ does not exist in $\mathcal{L}$, create it,
                        with links to any immediate parent and children nodes if such exist.
                Add $a(\bar{x})$ to node N.
                $\forall$ children nodes n_c of N
                        $\forall a_c \in$ PROJECTION $(n_c, \bar{x})$
                                Add a_c to N

Figure 3.12: Algorithm for determining supplementary axioms

Interstitial Axioms An axiom lattice of a domain class that includes only the minimal and supplementary axioms is generally very sparse. A user query may request attributes that are not associated with any node in such lattice. Thus a new node and axioms will have to be generated. Since the total number of nodes that could be computed is the power set of the attributes, the interstitial axioms and nodes are not exhaustively precomputed. Instead, they are derived, on demand, from other axioms already present in the lattice. The growth of the lattice results from user queries for unseen sets of attributes.

For example, suppose a query requesting large-seaport(cn cr pn) is received. If a node in the lattice for large-seaport corresponding to the set {cn, cr, pn} exists, the axioms cached in the node are returned. Otherwise, a new node for {cn, cr, pn} will be created and the corresponding interstitial axioms generated. To do so, all nodes which minimally contain the set {cn, cr, pn} become the child nodes of the new node. The axioms from these child nodes are projected into {cn, cr, pn} and used to populate the interstitial node. If no nodes contain the requested attribute set, there are no sources available for those attributes and the user query is unsatisfiable. Figure 3.13 shows the axiom lattice for large-seaport after a request for large-seaport(cn cr pn) has been satisfied and the interstitial node and axioms (marked with a hash - #) have been added. The algorithm for on-demand synthesis of interstitial axioms is shown in Figure 3.14.

27

{GC}
S3(S3.GC)
* S4(S4.GC) v S5(S5.GC)
* S1(S1.GC S1.CR) ∧ S1.CR > 7
* S1(S1.GC S1.PN) ∧ S2(S2.PN)
* S1(S1.GC S1.PN) ∧ S7(S7.PN)

{PN}
S2(S2.PN)
* S7(S7.PN)
* S4(S4.PN) v S5(S5.PN)
* S1(S1.CR S1.PN) ∧ S1.CR > 7
* S1(S1.GC S1.PN) ∧ S3(S3.GC)

{CN, PN}
S7(S7.CN S7.PN)

{CR, GC, PN}
S4(S4.CR S4.GC S4.PN) v S5(S5.CR S5.GC S5.PN)
S1(S1.CR S1.GC S1.PN) ∧ S1.CR > 7
S1(S1.CR S1.GC S1.PN) ∧ S2(S2.PN)
S1(S1.CR S1.GC S1.PN) ∧ S3(S3.GC)
* S1(S1.CR S1.GC S1.PN) ∧ S7(S7.PN)

{CN, CR, PN}
S1(S1.CR S1.PN) ∧ S7(S7.CN S7.PN)
[S4(S4.CR S4.PN) ∧ S7(S7.CN S7.PN)] v
  [S5(S5.CR S5.PN) ∧ S7(S7.CN S7.PN)]

{CN, CR, GC, PN}
S1(S1.CR S1.GC S1.PN) ∧ S7(S7.CN S7.PN)
[S4(S4.CR S4.GC S4.PN) ∧ S7(S7.CN S7.PN) v
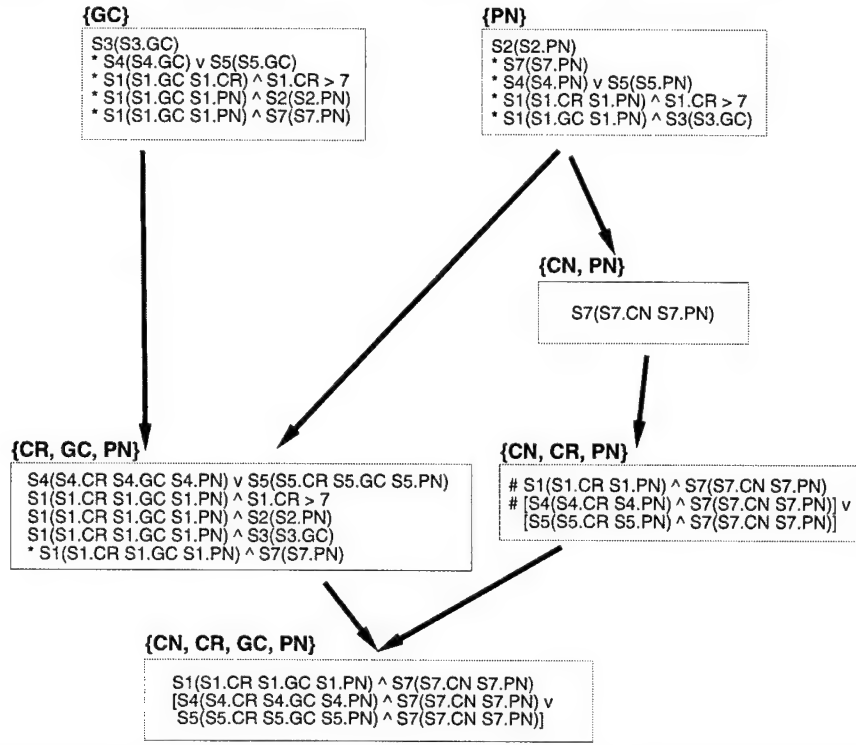 S5(S5.CR S5.GC S5.PN) ∧ S7(S7.CN S7.PN)]

Figure 3.13: Axiom lattice for large-seaport after generation of all supplementary axioms and single interstitial nodes for large-seaport(cn cr pn). Supplementary axioms are marked with an asterisk (*) and Interstitial axioms are marked with a hash (#).

Given some class C and set of attributes $\bar{x}$
Let $\mathcal{L}$ be the lattice for C.
If a node N providing $\bar{x}$ does not exist in $\mathcal{L}$,
    Let N_c be the set of nodes from $\mathcal{L}$ which correspond to a superset of $\bar{x}$.
    If $N_c = \emptyset$
        then fail ($\bar{x}$ for C cannot be retrieved from the available sources)
        else create N in $\mathcal{L}$ providing $\bar{x}$.
    Remove from N_c any nodes which dominate other nodes (set containment)
    $\forall$ immediate children $n_c \in N_c$
        $\forall a_c \in \text{PROJECTION}(n_c, \bar{x})$
            Add a_c to N

Figure 3.14: Algorithm for computing interstitial axioms

This section has presented an approach to source selection by pre-compiling a set of domain axioms and efficiently using them during query processing. These axioms can be used to improve the efficiency of a query planner by having readily available all the alternative ways of obtaining data for domain classes. Moreover, queries for which there are no sources are detected immediately.

4   Experimental Results

In this section we present some experimental results on the performance of our axiom compilation algorithm for real world knowledge bases and we compare the performance of source selection with and without the use of the compiled domain axioms. We tested the algorithms with domains independently built for the SIMS mediator system.

    Axiom compilation is fast in practice. In a transportation domain consisting of a domain model with 232 domain classes and 82 source classes distributed over 8 sources, the system compiled 77 axioms in 2.5 seconds. The example domain we have used throughout the chapter generates the 20 axioms from its 5 domain and 6 source classes in less than 0.1 seconds. Note that

this domain though small combines all of representational features in one hierarchy. These results suggest that compilation for domain of realistic size is efficient. Since the algorithm is incremental, even if the domain changes frequently the updates to the axioms can be performed efficiently.

In order to show the benefits of the compiled axioms during query processing, in the transportation domain we compared a version of the SIMS mediator that performs source selection using only the direct source descriptions with one that uses the compiled axioms. We tested both configurations with a set of 35 domain queries. In order to measure the complexity of rewriting for each query, we counted the number of reformulation operations that occur in the plans generated by the axiom-less version of SIMS. These reformulations are rewrites required to translate a domain-level query into a source-level query.

The results for the transportation domain are shown in Figure 3.15. Each point is the average time to rewrite all queries at the given level of difficulty. For the simple queries both systems perform comparably, but for harder queries the axiom-based version of SIMS is up to 60 times faster than the axiom-less version. These results confirm our hypothesis that using compiled axioms can significantly improve the scalability of mediator systems, while still providing a rich representation language for describing the contents of sources.
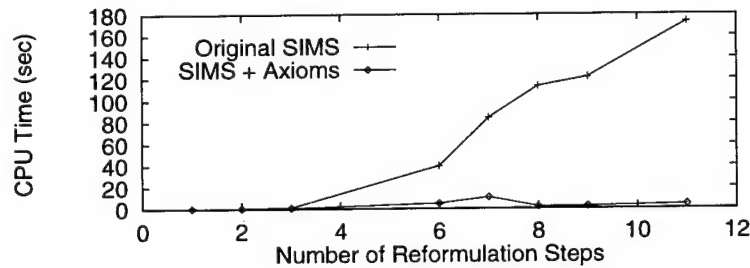

Figure 3.15: Comparison of the SIMS mediator with and without domain axiom compilation

5 Related Work

Heterogeneous multidatabase systems typically use a global domain model to provide the "glue" to integrate multiple data sources. The global domain model can be seen as providing common semantics to the information sources by means of views that relate terms in the sources with terms in the domain model. There are two ways of specifying these views, which have complementary properties [68]. The first approach, which has been widely used, is to integrate the sources by defining the global schema as a collection of views (queries) over the sources. This general approach has been used in a variety of systems, including Multibase [43], Pegasus [2], TSIMMIS [31], and HERMES [1]. An advantage is that the query rewriting algorithms are very efficient. The rewriting consists of substituting domain terms by their definitions, and simplifying the resulting source-level queries. A disadvantage is that adding or modifying a source can be quite difficult: all definitions in which that source appears have to be manually reconsidered.

The second approach, exemplified by the Information Manifold [45], is to define each data source as a view of the global domain model. Specifically, each source predicate is defined as a view over domain predicates. An advantage of this method is that modifying the definitions or adding new sources is quite straightforward because each source is defined independently from others. A disadvantage is that the algorithms to rewrite a domain-level query into a source-level query involves testing containment of views, which is computationally expensive [45, 44]. The work presented in this chapter combines the best of both approaches. Initially, sources are conveniently defined in domain terms, so that new sources are easily incorporated or updated. Then, axioms defining domain terms as views over source terms are automatically compiled, so that query processing can be performed more efficiently.

Another recent approach that also attempts to combine some of the features of both approaches is illustrated by the Garlic system [30, 62]. Instead of defining views of the sources or of the domain model, Garlic maintains a list of the sources that can provide a portion of the data for each class and when it receives a query it queries each of the possibly relevant sources to determine which portion of the required data the sources can provide. An advantage of this approach is that the individual sources can provide accurate estimates of the cost of retrieving the data, so Garlic can put together efficient plans. The work on Garlic addresses a different problem than the one addressed in this chapter and could be usefully combined with our approach. Our work focuses on providing a rich representation language for describing the contents of sources and efficiently determining how those sources can be combined to answer a query. In contrast, Garlic focuses on finding the most efficient combination of possible sources, assuming a very simple representation language. The two approaches could be combined by first using our work to represent the sources and determine how to combine the relevant sources and then using the Garlic approach to estimate the cost of the different plans and select the most efficient one.

The Information Manifold (IM) [45, 44] also provides a rich representation languages that is combination of datalog and description logic. However, view rewriting in its language is intractable and can become undecidable in the presence of recursion [46, 47]. Nevertheless, Levy argues that his algorithm focuses on the relevant information sources, which is small in practice, and the size of typical queries will also be small, so the theoretical intractability does not present a major problem. Views in IM express containment relationships and IM produces query plans that are maximally contained rewritings. This implies that the answer may be incomplete but the system does not signal it. View definitions in SIMS express equality and its query plans

29

provide complete answers. SIMS can straightforwardly determine when a user query cannot be answered. More importantly, SIMS can point to which classes of information it is missing and guide query relaxation.

Perhaps the most general approach to information integration is given by context logic [13, 29] which extends the predicate calculus with a new modality, (ist c ϕ), meaning that a logical sentence ϕ is true in a context c. The logic is sound and complete but undecidable. Lifting axioms relate formulas in different contexts, similarly to the view definitions above. The Carnot system [14], and its successor InfoSleuth [35], use restricted forms of lifting axioms, similar to those of the Multibase approach, expressed in a frame-based common language. A recent system using full context logic is [23]. Another general system is Infomaster [17], which treats view rewriting as a form of abduction and uses a model elimination theorem prover to implement it. Rather than focus solely on generality, our work considers also efficient query planning techniques, such as the domain precompilation presented in this chapter, while at the same being able to represent a great variety of sources in practice.

6   Contributions and Future Work

We have presented an approach to integrating information from heterogeneous data sources that combines the *flexibility* of view rewriting with the *efficiency* of query processing typical of systems such as Multibase and TSIMMIS. In order to do so, our system allows the user to conveniently define the information sources in terms of the domain model, and automatically compile these source descriptions into a set of axioms that specify the domain model classes as formulas in source terms. Based on the axioms compiled off-line, the system computes at run-time the most appropriate rewriting for answering a query by simply instantiating the corresponding axioms. Our central idea is to shift the complexity of view rewriting to a preprocessing step that can be done off-line and will be amortized as the mediator processes user queries.

Our compiled axioms facilitate query processing in the presence of *partial information*. The modeling language allows the user to specify when a source has complete information for a class. However, in many real-world domains complete information will not be available and the queries will have to be answered based on the available partial information. Having axioms compiled for each domain class allows us to immediately recognize unsatisfiable queries and to identify exactly what class of information is missing and inform the user what information can be provided.

The compiled axioms also facilitate *replanning after failure*. Information sources in distributed heterogeneous environment may become unavailable during the execution of a query. It is desirable to provide an alternative way to answer that query, reusing parts of the executed plan if possible. These alternative ways of obtaining the required data are at hand in the compiled axioms.

We are in the process of extending our source descriptions and axiom compilation algorithm to include *binding patterns* [67]. Binding patterns represent a type of constraints on the capabilities of an information source, in which the source needs one or several input values in order to produce additional data. This type of behavior frequently occurs in Web sources. We are also developing a generic cost-based query planner based on rewriting rules that take advantage of the source selection techniques introduced here [3, 4].

Chapter 4

Planning by Rewriting: Efficiently Generating High-Quality Plans

1 Introduction

Planning is the process of generating a network of actions that achieves a desired goal from an initial state of the world. Domain independent planning accepts as input, not only the initial state and the goal, but also the domain specification (i.e., the operators). This is a problem of considerable practical significance, but domain-independent planning is computationally hard except for its simplest formulations [18]. Moreover, in many circumstances it is not enough to find any solution plan since the quality of the solution is important. This chapter presents a new paradigm for efficiently generating high-quality plans.

Two observations guided the present work. First, there are two sources of complexity in planning:

- Satisfiability: the difficulty of finding any solution to a planning problem.
- Optimization: the difficulty of finding the optimal solution according to a given cost metric.

For a given domain, each of these facets may contribute differently to the complexity of planning. In particular, there are many domains in which the satisfiability problem is easy and their complexity is dominated by the optimization problem. For example, there may be many plans that would solve the problem, so finding one is simple (that is, in polynomial time), but the cost of each solution varies greatly so that finding the optimal one may be difficult. We shall refer to these domains as optimization domains. Some optimization domains of great practical interest are query access planning and process planning.[1]

Second, planning problems have a great deal of structure. Plans are a type of graphs with strong semantics, determined both by the general properties of planning and each particular domain specification. This structure should and can be exploited to improve the efficiency of the planning process.

Prompted by the previous observations, we developed a novel approach for efficient planning in optimization domains: Planning by Rewriting (PBR). The framework works in two phases:

1. Generate an initial solution plan. Recall, that in optimization domains this is easy. However, the quality of this initial plan may be far from optimal.

2. Iteratively rewrite the current solution plan improving its quality using a set of plan rewriting rules until either an acceptable solution is found or a resource limit is reached.

There are several important points to note in this basic framework. First, the rewritten plans are always solutions to the given planning problem. Thus, the search occurs in the space of solution plans, which is in many cases much smaller than the space of partial plans that other planning systems usually explore. Second, efficient search of the space of rewritings is critical to the success of the method. Thus, we adapt techniques from local search to help in this process. Finally, our framework yields an anytime algorithm [16]. The planner always has a solution to offer at any point in its computation (modulo the initial plan generation, which should be fast). This is a clear advantage over traditional planning approaches, which must run to completion before producing a solution. Thus, our system allows the possibility of trading off planning effort and plan quality. For example, in query planning the quality of a plan is its execution time and it may not make sense to keep planning if the cost of the current plan is small enough, even if a cheaper one could be found.

As motivation, consider two domains: query processing in a distributed, heterogeneous environment and manufacturing process planning. Distributed query processing [75] involves generating a plan that efficiently computes a user query. This plan is composed of data retrieval actions at diverse information sources and operations on this data (such as join, selection, etc). Some systems use a general-purpose planner to solve this problem [40]. In this domain it is relatively easy to construct an initial plan and then transform it using a gradient-descent search to reduce its cost. The plan transformations exploit the commutative and associative properties of the (relational algebra) operators and facts such as that when a group of operators

---

[1] Interestingly, one of the most widely studied planning domains, the blocksworld, also has this property.

can be executed together at a remote information source it is generally more efficient to do so. Figure 4.1 shows some sample transformations.

```
join-swap
```
$get(q1, db1) \bowtie (get(q2, db2) \bowtie get(q3, db3)) \Leftrightarrow$
$get(q2, db2) \bowtie (get(q1, db1) \bowtie get(q3, db3))$
```
remote-join-eval
```
$(get(R, db) \bowtie get(S, db)) \wedge capability(db, join) \Rightarrow get(R \bowtie S, db)$

Figure 4.1: Transformations in Query Planning

In manufacturing, the problem is to find an economical plan of machining operations that implement the desired features of a design. In a feature-based approach [54] it is possible to enumerate the possible actions involved in building a piece by analyzing its CAD model. It is more difficult to find an ordering of the operations and the setups that optimize the machining cost. However, similar to query planning, it is possible to incrementally transform a (possibly inefficient) initial plan. Often, the order of actions does not affect the design goal, only the quality of the plan, thus actions can commute. Also, it is important to minimize the number of setups because fixing a piece on a machine is a rather time consuming operation. Such grouping of machining operations on a setup is analogous to evaluating a subquery at a remote information source.

In summary, this chapter develops a new planning paradigm yielding several contributions. First, by using local search techniques, high-quality plans can be efficiently generated. Second, the rewriting rules provide a natural and convenient mechanism to specify complex plan transformations. Third, it offers a new anytime planning algorithm.

2 Planning by Rewriting

We will describe the main issues in Planning by Rewriting as an instantiation of the local search idea [56]:

- *Selection of an initial feasible point*: How to efficiently generate an initial solution plan.

- *Generation of a local neighborhood*: The neighborhood is the set of plans obtained from the application of the plan rewriting rules.

- *Cost function to minimize*: The given plan evaluation function could range from a simple domain independent cost metric, such as the number of steps, to more complex domain specific ones, such as query evaluation cost or manufacturing time for a set of parts.

- *Selection of the next point*: What is the next plan to consider. This choice determines how the global space will be explored and has a significant impact on the efficiency of planning. For example, steepest descent, first improvement, random walk, etc.

In the following subsections we expand these topics. First, we introduce some background on planning and rewriting. Second, we discuss the initial plan generation. Third, we show how the local neighborhood is generated by the rewriting rules and present their syntax, their semantics, and a rule taxonomy. Finally, we address the selection of the next plan.

2.1 Planning and Rewriting Concepts

A plan is represented by a graph notation, in the spirit of partial-order causal-link (POCL) planners, such as UCPOP [58]. The nodes are plan steps, that is, domain actions. The edges specify a temporal ordering relation among steps, imposed by causal links and ordering constraints. A causal link is a record of how a condition is used in a plan. This record contains the condition, a step that produces (establishes) it, and a step that consumes it (that is, a step which needs it as a precondition). By causality, the producer must precede the consumer. The ordering constraints arise from solving operator threats and resource conflicts. An operator threat occurs when a step has an effect that negates the condition of a causal link and can possibly be ordered between its producer and its consumer. To prevent this situation, which possibly makes the plan inconsistent, POCL planners order the threatening step either before the producer (promotion) or after the consumer (demotion).

Operators may need to use certain resources to perform their actions. In this chapter, we consider unit non-consumable resources, that is, those that are fully acquired by an operator until the completion of its action, and then released to be reused [38]. For this type of resource, steps requiring the same resource have to be sequentially ordered. Finally, note that all conditions in the plan are fully ground because we start with a complete initial plan.

A plan rewriting rule, akin to term and graph rewriting rules, specifies the replacement under certain conditions of a partial plan by another partial plan. Our system ensures that the rewritten plan remains complete and consistent. These rules are intended to improve the quality of the plans.

2.2 Generation of an Initial Plan

Fast initial plan generation is domain-specific in nature. It requires the user to specify an efficient mechanism to compute the initial solution plan. By the definition of optimization domains this should not be hard. We have experimented with two approaches to construct feasible initial plans: using a planner with search control rules and exploiting simple domain-specific approximation algorithms.

A very general way of efficiently constructing plans is to use a domain-independent generative planner that accepts search control rules. By setting the type of search and providing a strong bias by means of the search control rules, the planner can quickly generate a valid, although possibly suboptimal, initial plan. For example, in the manufacturing domain we used depth-first search and a goal selection heuristic based on abstraction hierarchies [37]. This combination quickly generates a feasible plan, but often the time required to manufacture all objects is suboptimal.

For many domains, we expect that simple domain-dependent greedy algorithms will provide good initial plans. For example, in the query planning domain, the system can easily generate initial query evaluation plans by parsing the given query. In the blocksworld it is also straightforward to generate a solution in linear time using the naive algorithm: put all blocks on the table and build the desired towers from the bottom up.

2.3 Generation of a Local Neighborhood

The plan rewriting rules determine the neighborhood of the current plan to be explored. They embody the domain-specific knowledge about what transformations of a solution plan are likely to result in higher-quality solutions. In this section we describe the syntax and the semantics of the rules, as well a taxonomy of plan rewriting rules.

Rule Syntax and Semantics

First, we introduce the rule syntax and semantics through some examples. Then, we provide a formal description. A sample rule in the blocks world domain is given in Figure 4.2. Intuitively, it says that, whenever possible, it is better to stack a block on top of another directly, rather than first moving it to the table.

```
(define-rule :name avoid-move-twice
  :if (:operators ((?n1 (unstack ?b1 ?b2))
                   (?n2 (stack ?b1 ?b3 Table)))
       :links (?n1 (on ?b1 Table) ?n2)
       :constraints ((possibly-adjacent ?n1 ?n2)
                     (:neq ?b2 ?b3)))
  :replace (:operators (?n1 ?n2))
  :with (:operators (?n3 (stack ?b1 ?b3 ?b2))))
```

Figure 4.2: Blocks World Rewriting Rule

A rule for a manufacturing domain [51] is shown in Figure 4.3. It states that if a plan includes two consecutive punching operations to make holes in two different objects, but another machine, a drill-press, is also available, the plan can be parallelized by replacing one of the punch operations by using the drill-press.

```
(define-rule :name punch-by-drill-press
  :if (:operators ((?n1 (punch ?o1 ?width1 ?orn1))
                   (?n2 (punch ?o2 ?width2 ?orn2)))
       :links (?n1 ?n2)
       :constraints ((:neq ?o1 ?o2)
                     (possibly-adjacent ?n1 ?n2)))
  :replace (:operators (?n1))
  :with (:operators
           (?n3 (drill-press ?o1 ?width1 ?orn1))))
```

Figure 4.3: Process Planning Rewriting Rule

In general, the rule syntax follows the template in Figure 4.4. The rewriting algorithm is outlined in Figure 4.5. The semantics of the rules is as follows. The antecedent, the :if field, describes a graph specification (operators, links, and constraints) that is matched against the plan. The :operators field consists of a list of step number and step predicate pairs. Each step predicate is interpreted as an step action (or as one of the resources used by the step, if the keyword :resource is present, e.g. Figure 4.7). The :links field consists of a list of link specifications. A link specification can match either any ordering link in the plan, a causal link if a predicate is given, or an ordering link introduced in the resolution of threats (if the keyword :threat is present). Finally, built-in and user-defined predicates can be specified in the :constraints field. The built-in predicates include inequalities (:neq), comparison, and arithmetic predicates. The user-defined predicates may act as filters on the previous variables or introduce new variables (and compute new values for them). Formally, the language of the antecedent forms a conjunctive query with interpreted predicates against the plan graph. The rule matches can be computed either all at the same time, as in bottom-up evaluation of logic databases, or one-at-a-time as in Prolog. Which option is preferable depends on the search strategy.

33

```
(define-rule :name <rule-name>
 :if (:operators (((<nv> <np> {:resource}) ...)
     :links (((<nv> {<lp>|:threat} <nv>) ...)
     :constraints (<ip> ...))
 :replace (:operators (<nv> ...)
          :links (((<nv> {<lp>|:threat} <nv>) ...))
 :with (:operators (((<nv> <np> {:resource}) ...)
       :links (((<nv> {<lp>} <nv>) ...)))

<nv> = node variable, <np> = node predicate,
<lp> = causal link predicate, {} = optional
<ip> = interpreted predicate,  | = alternative
```

Figure 4.4: Rewriting Rule Template

1. Match rule antecedent, :if field, against the plan, returning a set of candidate rule instantiations.

2. For each antecedent instantiation:

    (a) Remove from the plan the subgraph specified in the :replace field.

    (b) Generate all *consistent* embeddings of the subgraph specified in the :with field.

Figure 4.5: Outline of Plan Rewriting Algorithm

Rules must be safe, that is, all the variables appearing in the consequent of the rules, :replace and :with fields, have to appear in the antecedent. The :replace field identifies the subgraph that is going to be removed from the plan (a subset of steps and links of the antecedent). The :with field specifies the replacement subgraph. The system generates all valid embeddings of the replacement subplan into the original plan (once the subplan in the :replace field has been removed). Thus, a single rule instantiation may produce several rewritten plans. The formal conditions for valid rewriting, a generalization from plan merging in [26], are shown in Figure 4.6. It is possible to define rules whose application provably yields a correct plan. However, this *eager* approach would require the generation of many rules with very long and specific antecedents, which are possibly expensive to match. An alternative is a *lazy* approach in which the rule antecedents only include a subset of the conditions necessary for a valid rewriting. In this case, when the rules are applied, the rewritten plans are checked for correctness. The "lazy" approach allows the specification of more natural rules that express the main idea of the transformation instead of focusing on technicalities or rare cases. We used the latter for our experiments.

A subplan $S1$, embedded in a plan P, can be replaced by a subplan $S2$, resulting in plan P', iff
there exists an ordering O, such that
$P' = (P - S1) \cup S2 \cup O$ is a consistent plan, and $NetPreconditions(S2, P') \subseteq NetPreconditions(S1, P)$, and $UsefulEffects(S1, P) \subseteq UsefulEffects(S2, P')$.

Useful Effects of a subplan S, embedded in a plan P, are those conditions present in causal links whose producer is in S and whose consumer is in $P - S$.

Net Preconditions of a subplan S, embedded in a plan P, are those conditions in causal links whose consumer is in S and whose producer is in $P - S$.

Figure 4.6: Conditions for Valid Rewriting

A Taxonomy of Plan Rewriting Rules

In order to guide the user in defining plan rewriting rules for a domain or to help in designing algorithms that may automatically deduce the rules from the domain specification (see Future Work), it is helpful to know what kinds of rules are useful. So far we have identified the following general types of transformation rules.

Reorder: These are rules based on algebraic properties of the operators, such as commutative, associative and distributive laws. For example, the commutative rule that reorders two operators that need the same resource in Figure 4.7, and the join-swap rule in Figure 4.1 that combines the commutative and associative properties of the relational algebra.

Collapse: These are rules that replace a subplan by a smaller subplan. For example, when several operators can be replaced by one, as in the remote-join-eval rule in Figure 4.1, which prefers to evaluate a join between two tables that come from the same source at the remote source rather than locally (if the source has join processing capabilities). Another example is the blocksworld rule in Figure 4.2.

```
(define-rule :name resource-swap
  :if (:operators (((?n1 (machine ?x) :resource)
                    (?n2 (machine ?x) :resource))
      :links ((?n1 :threat ?n2)))
  :replace (:links (?n1 ?n2))
  :with (:links (?n2 ?n1)))
```

Figure 4.7: Reorder Rewriting Rule

Expand: These are rules that do the inverse of collapse. Although we did not find this rule type in the domains analyzed so far, it is easy to imagine a situation in which an expensive operator can be replaced by a set of operators that are cheaper as a whole. For example, when some of these operators are already present in the plan and can be synergistically reused.

Parallelize: These are rules that replace a subplan with an equivalent alternative subplan that requires fewer ordering constraints. A typical case is when there are redundant or alternative resources that the operators can use. For example, the rule punch-by-drill-press in Figure 4.3.

2.4   Selection of Next Plan

The strategy to select the next plan to consider determines the way the solution space is searched. The rules generate the "natural perturbations" of a plan, but which rewriting, if any, will lead towards the global optimum cannot be predicted in general. We have explored gradient descent techniques, such as first improvement and steepest descent. In *first improvement*, the next plan to consider is the first rewriting that improves the cost. This has the advantage that the neighborhood is generated only up to the point such a plan is found, but the improvement may not be the best that could be achieved in that neighborhood. In *steepest descent*, the minimum cost plan within the neighborhood is chosen. This guarantees the biggest improvement in cost in each iteration, but it requires the whole neighborhood to be searched.

In general, the space of rewritings and the cost functions are not convex, thus our gradient descent techniques can get caught in local minima. To move towards the optimum escaping low-quality local minima, we used two techniques: restart and random walk. In the first one, the system restarts the rewriting process a fixed number of times from a different initial plan. This technique requires an initial plan generator that is able to provide several different/random initial plans. The second technique is applied when the local minima are not strict and consists of a random walk of a fixed length along the plateau.

3   Initial Results

We have implemented the planner described in this chapter and applied it in several different application domains. In this section we report on our initial results in the domains of manufacturing process planning and distributed query planning.

3.1   Process planning

The task in the manufacturing process planning domain is to find a plan to manufacture a set of parts. We implemented the domain specification in [51]. This domain contains a variety of machines, such as a lathe, punch, spray painter, welder, etc, which are used to perform various operations to produce a set of parts. In this domain all of the machining operations are assumed to take unit time and the optimal plan is the one that requires the minimal length schedule. There are ten possible machining operations for making a part. Sample rewriting rules for this domain appear in Figures 4.3 and 4.7.

To evaluate the performance of Planning by Rewriting (PBR), we compared it to a planner called Sage [39], which is an extension of UCPOP that supports resources, execution and replanning. For PBR, we defined ten plan rewriting rules for this domain and used a steepest descent search. We ran Sage with a best-first search over the length of the schedule (Sage-BFS) in order to find the optimal plan. We used Sage with depth-first search and a goal selection heuristic based on abstraction hierarchies (Sage-DFS) to generate plans as fast as possible. Sage-DFS is also used for the initial plan generator for PBR. We tested each of the three systems on 300 problems that ranged from 1 to 12 goals (25 in each set). There were 80 provably unsolvable problems. Sage-DFS was able to solve 22 more problems than Sage-BFS in the given search limit of 50,000 nodes (Sage-DFS proved 16 solvable and 6 unsolvable). The plan size (number of steps) grows linearly with the number of goals, from 3 steps for the one goal problems to 14 steps for the 12 goal problems.

The results are shown in Figures 4.8 and 4.9. Figure 4.8 shows the average time on the solvable problems for each problem set in the three configurations. Figure 4.9 shows the average schedule length for the problems solved by all planners. As shown in the graphs, PBR takes slightly longer than Sage-DFS, as expected since Sage-DFS is also used to generate the initial plans for PBR, but is able to improve their quality significantly. PBR performs about the same as Sage-BFS on the easy problems, both in time and in quality. For harder problems, PBR is much more efficient than Sage-BFS and it produces plans whose quality is close to the optimal. These results show the benefits of finding an suboptimal initial plan quickly and then efficiently transforming it to improve its quality as proposed in PBR.
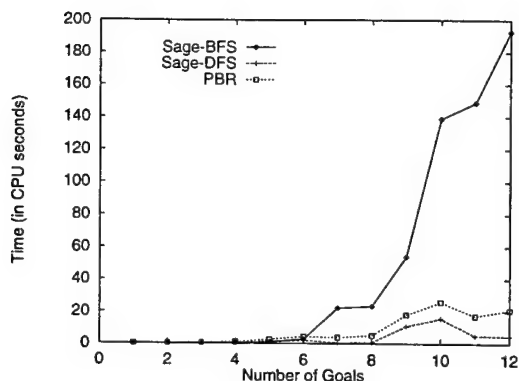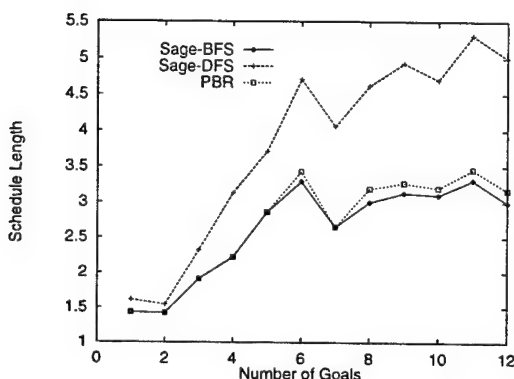
35

Figure 4.8: Manufacturing Time Comparison

Figure 4.9: Schedule Length Comparison

3.2 Query planning

Distributed query processing involves generating a plan that efficiently computes a user query. This plan is composed of data retrieval actions at diverse information sources and operations on this data. We used a simplified domain for the query planner (Sage) of the SIMS mediator [9]. We compare the performance and quality of the Sage planner and PBR for this domain, where the query plans are trees of join operations.

In the query planning domain, Sage performs a best-first search with a heuristic commonly used in query optimization that explores only the space of left join trees (Sage-BFS). For PBR, we defined the join-swap rule of Figure 4.1. The initial plans were random depth-first search parses of the query (Sage-DFS). To escape local minima, PBR generates and rewrites three random initial plans and picks the best rewriting. The cost metric for all planners is based on an estimation of the cost of the join operations and the size of the intermediate results transmitted between the sources and the mediator. In this experiment, we used a set of 43 conjunctive queries previously defined for a logistics planning application involving from one to seven relations. All queries could be solved by all planners. A set of eight relation joins could not be solved by Sage within 50,000 nodes, while PBR could easily solve them with low cost plans.

The results are shown in Figures 4.10, 4.11, and 4.12. Figure 4.10 shows the average time for each query set for query sizes from one to seven. The times for PBR includes both the generation of the three random initial plans and their rewriting. Figure 4.11 shows the average quality of Sage-DFS and PBR normalized with respect to Sage-BFS. The normalization is done for clarity because the values for Sage-BFS query cost vary considerably and we want to show how PBR and Sage-DFS perform relative to Sage-BFS as the problem size increases. The graph shows that Sage-DFS produces very poor quality plans on the large problems. Figure 4.12 shows in more detail the average quality of PBR normalized with respect to Sage-BFS. This graph shows that PBR performs as well as Sage-BFS for the smaller queries and even finds better solutions for the larger ones (up to 12% better). This is not surprising because of the restricted space of left trees that Sage-BFS is searching. As in the manufacturing domain, PBR shows better scaling properties than the corresponding systematic algorithms.

4 Related Work

Some of the most closely related work is on plan merging [26]. Their system solves a complex goal by dividing it into subgoals, solving the subproblems, and combining the partial solutions exploiting synergies. They improve the quality of a plan by replacing a set of operators by one operator that can do the same job. Planning by Rewriting differs in that it starts with
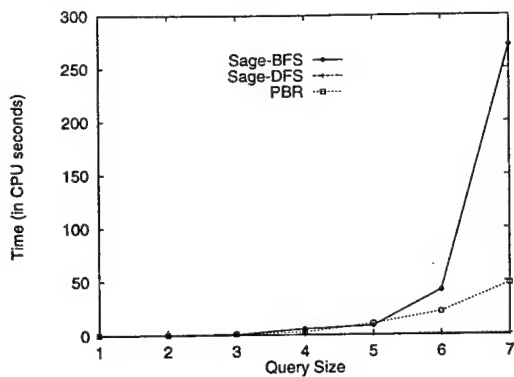
Figure 4.10: Query Planning Time Comparison
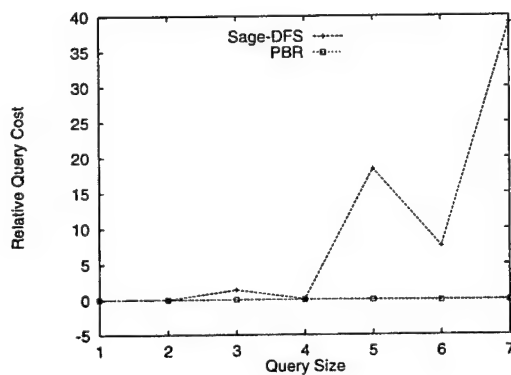
Figure 4.11: Query Plan Quality Comparison

a complete solution plan for the original goal, and it generalizes plan merging by allowing the replacement of a subplan by another subplan, thus expanding the types of plan transformations and the opportunities for cost reduction.

Case-based planning also attempts to solve a problem by modifying a previous solution [69, 57]. Systematic algorithms, such as [33], invert the decisions done in refinement planning to find a path between the solution to a similar old problem and the new problem. Our work modifies a solution to the current problem, so there is no need for similarity metrics, nor retrieval process. Moreover, our rewriting rules indicate how to transform a solution into another solution plan, rather than searching blindly up and down the space of partial plans. However, the rules in PBR may search the space of rewritings non-systematically. Such an effect is ameliorated by the gradient-descent search strategy.

Local search has a long tradition in combinatorial optimization [56]. Local improvement ideas have found application in constraint satisfaction, scheduling, and heuristic search. In constraint satisfaction, [52] start with a complete, but inconsistent,

Figure 4.12: Query Plan Quality (PBR only)

37

variable assignment and efficiently search the space of repairs using a simple heuristic, min-conflicts. In our work we focus on a STRIPS-like planning paradigm (with fairly expressive operators) in which the rewritings yield complete and consistent plans, as opposed to complete but inconsistent variable assignments. In work on scheduling and rescheduling, [76] define a set of general, but fixed, repairs methods, and use simulated annealing to search the space of schedules. Our plans are networks of actions as opposed to the metric-tim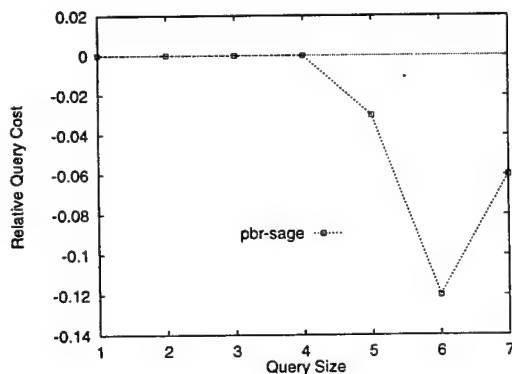e total-order tasks in that work. Also we can easily specify different rewriting rules (general or specific) to suit each domain, as opposed to their fixed strategies. Related ideas have been used in heuristic search [60]. In that work, first they find a valid sequence of operators using an approximate algorithm. Then, they identify segments of this sequence, take their initial and end states, and heuristically search for a shorter path for that segment (the cost metric is the path length). They are doing a state-space search, while PBR is doing a plan-space search. The least-committed partial-order nature of PBR allows it to optimize the plans in ways that cannot be achieved by optimizing linear subsequences.

A variety of research has attacked the complexity of planning. Some systems incorporate automatically learned search control, for example, search control rules [51] and abstraction [37]. Our system does not learn the rewriting rules currently (see Future Work). Other work has reduced planning to propositional satisfiability, which can be solved by stochastic local search [36]. These approaches do not specifically address plan quality, or else they consider only very simple cost metrics (such as the number of steps). Quality-improving control rules are learned in [59], but planning efficiency was not significantly improved. By exploiting domain-specific knowledge, conveniently expressed as plan rewriting rules, and the local search approach, we improve both plan efficiency and quality. Moreover, we provide an anytime algorithm while other approaches must run to completion.

Some domain specific planners have also used a transformational approach, for example, query evaluation in centralized databases [28]. They parse the query to obtain an initial evaluation plan and iteratively transform this plan using a set of rules based on the algebra of the data model. PBR offers a more general and easily extensible framework to tackle more complex information gathering domains. Finally, the research in graph rewriting [64] may provide efficient matching algorithms and perhaps another implementation vehicle using high-level graph-rewriting programming languages.

5   Future Work

There are several issues that we plan to address more thoroughly in the future: initial plan generation, automatic rule generation, and alternative search strategies. Initial plan generation is domain-specific, but we intend to provide a domain independent procedural plan construction language to allow the convenient specification of plan construction algorithms. It will include primitives for adding steps and ordering constraints, but it will hide the complexity of the data structures used to represent the actual plans.

We believe that the rules can be generated by fully automated procedures in many domains. The methods can range from static analysis of the domain operators to analysis of sample equivalent plans (that achieve the same goals but at different costs). Note the similarity with methods to automatically deduce search control [51, 19] and also the need to deal with the utility problem.

There are many techniques in the local search literature that we could adapt to our framework. In particular, we plan to explore variable depth rewriting, and variations of tabu search. In *variable depth*, a sequence of rewritings is applied in order to overcome initial cost increases that eventually would lead to strong cost reductions. This idea leads to the creation of rule programs, which specify how a set of rules are applied to the plan, possibly depending on run-time conditions. In *tabu search*, some of the rewritings are temporarily forbidden regardless of their cost. This is useful to avoid returning to some previously visited plan and thus cycling. Also, it forces the search not to be concentrated in a small local area around a local minimum. Finally, we plan to improve the planner implementation. For example, a RETE-like graph matcher [25] would make the system much more efficient.

6   Conclusions

We presented a new paradigm for efficient high-quality planning based on local search and plan rewriting, and we provided initial experimental support for its usefulness in several domains. This framework achieves a balance between domain knowledge, conveniently expressed as plan rewriting rules, and general local search techniques that have been proved useful in many hard combinatorial problems. We expect that these ideas will push the frontier of solvable problems for many domains into the range of real-world problems in which high quality plans and anytime behavior are needed.

Chapter 5

Planning, Executing, Sensing, and Replanning

1   Introduction

The task of information gathering requires locating, retrieving, and integrating information from large numbers of distributed and heterogeneous information sources. In this environment, flexibility and efficiency are critical. The usual approach of generating a static plan for processing information and then executing it is inflexible and may be very inefficient if problems arise during query processing. The problem is that there may be many information sources from which to choose, actions may fail, the system has incomplete knowledge about the available information, and new goals may arise at any time.

To address these problems, we have developed a planning system that builds on previous work on planning, execution, sensing, and replanning. The planner, which we call Sage, was implemented by augmenting UCPOP [58, 10] with the capabilities to produce parallel execution plans [72, 38], interleave planning and execution [5, 20], support run-time variables for sensing [5, 21], perform replanning where appropriate, and plan for new goals as they arise. We have integrated all of these capabilities into a single, unified system in which planning, execution, sensing, and replanning can be performed during execution. This allows the system to replan portions of the plan that is currently being executed, receive and plan new tasks within the context of the executing plan, and interleave sensing actions with planning in order to improve efficiency.

Before describing the integration of planning and execution, we first describe the information gathering task and how it can be cast as a planning problem in a general planning framework (Section 2). Next, we present our approach to tightly integrating planning and execution (Section 3). This integration is used to support planning for new goals, replanning for failure, and the interleaving of sensing actions to gather additional information for planning (Section 4). We compare this work to previous work in planning as well as information gathering and query processing (Section 5). Finally, we conclude with a discussion of the contributions of the chapter (Section 6).

2   Planning for Information Gathering

Information gathering requires selecting, integrating, and retrieving data from distributed and heterogeneous information sources in order to satisfy a query. The relevant data must be selected from numerous, possibly overlapping or replicated sources. Integrating the information may be costly, especially when combining data from different sites. Retrieving the information may be time consuming due to the distribution of data and the contention for limited resources.

To solve this problem, we have developed a planner called Sage that builds on the UCPOP partial-order planner [10]. UCPOP provides an expressive operator language that includes conjunction, negation, disjunction, existential and universal quantifiers, conditional effects, and a functional interface that allows preconditions to be implemented as Lisp functions. We extended this planner to support simultaneous action execution and to tightly integrate planning and execution. The execution is presented in the next section, and the support for simultaneous actions was previously addressed in [38] and will be briefly described here.

Partial-order planners, such as UCPOP, produce plans with actions that are unordered. However, if two actions are left unordered they can be executed in either order, but not simultaneously. To execute actions in parallel in a partial-order planner requires that (1) actions can be executed simultaneously without changing the outcome of the individual actions, and (2) any potential resource conflicts must be captured in the representation of the operators in order to avoid conflicts during execution. We assume that the first condition holds (as it does in the information gathering domain described below) and we extended the planner to support the second condition. To support reasoning about resources, we added an explicit resource declaration to the action language, which describes the resources required when executing an action. We also augmented the planner to identify and remove potential resource conflicts. With these extensions, any actions left unordered in the final plan can be executed simultaneously.

In the remainder of this section we describe how the information gathering task is cast as a planning problem in Sage. This problem requires producing a plan for generating a requested set of data. This involves selecting the sources for the data, the operations for processing the data, the sites where the operations will be performed, and the order in which to perform the

39

operations. Since data can be retrieved from multiple sources and the operations can be performed in a variety of orders, the space of possible plans is large.

An information gathering goal consists of a description of a set of desired data as well as the location where that data is to be sent. For example, Table 5.1 illustrates a goal which specifies that the set of data be sent to the OUTPUT device of the SIMS information mediator [7, 41]. The goal also specifies the data to be retrieved and is defined using the syntax of the query language of the Loom knowledge representation system [50]. This particular query requests all port names of seaports that are sufficiently deep to accommodate "breakbulk" ships.

```
(available output sims
    (retrieve (?port-name)
        (:and (seaport ?sport)
              (port-name ?sport ?port-name)
              (channel-of ?sport ?channel)
              (channel-depth ?channel ?depth)
              (transport-ship ?ship)
              (vehicle-type-name ?ship "breakbulk")
              (max-draft ?ship ?draft)
              (< ?draft ?depth))))
```

Table 5.1: An information gathering goal

The initial state of a problem defines the available information sources (e.g., databases) and the servers (e.g., an Oracle DBMS) they are running on. The example shown in Table 5.2 defines two servers, an Oracle database server running on an HP workstation, called hp-oracle, and an another Oracle server running on a Sun workstation, called sun-oracle. Both servers contain identical copies of the GEO and ASSETS databases. In addition to this information, a description of the contents of the information sources is stored in a Loom knowledge base. However, this information is static and is accessed directly through the functional interface rather than through the literals listed in the initial state.

```
((source-available geo hp-oracle)
 (source-available assets hp-oracle)
 (source-available geo sun-oracle)
 (source-available assets sun-oracle))
```

Table 5.2: An initial state

For this domain, Sage uses a set of ten general operators to plan out the processing of a query. They include a **move** operator for moving a set of data from one information source to another, a join operator that combines two sets of data into a combined set of data, and a **select-source** operator for selecting the information source for retrieving a set of data. The other operators perform additional processing of data (**select**, **compute**, and **assignment**) or reformulate queries using background knowledge (**generalize, specialize, definition,** and **decompose**). Each operator is instantiated at planning time with the particular set of data being manipulated as well as the database where the manipulation is being performed.

Consider the operator shown in Table 5.3, which defines a join performed in the local system. This operator is used to achieve the goal of making some information available in the local knowledge base of the SIMS information mediator. It does this by partitioning the request into two subsets of the requested data, retrieving that information into the local system, and then joining the data together to produce the requested set of data. The **available** preconditions are achieved by other operators and the **join-partition** precondition is defined by a function that produces the relevant partitions of the requested data.

This planning domain differs from many of the domains that previous planning work has focused on in two significant ways. First, there are few interactions between the operators. The main source of interaction arises in handling resource conflicts when two operators require access to the same server. Second, it is not sufficient to find any solution to a problem; the goal

```
(define (operator join)
  :parameters (?join-op ?data ?data-a ?data-b)
  :precondition
     (:and (join-partition ?data ?join-op
                            ?data-a ?data-b)
           (available local sims ?data-a)
           (available local sims ?data-b))
  :effect (available local sims ?data))
```
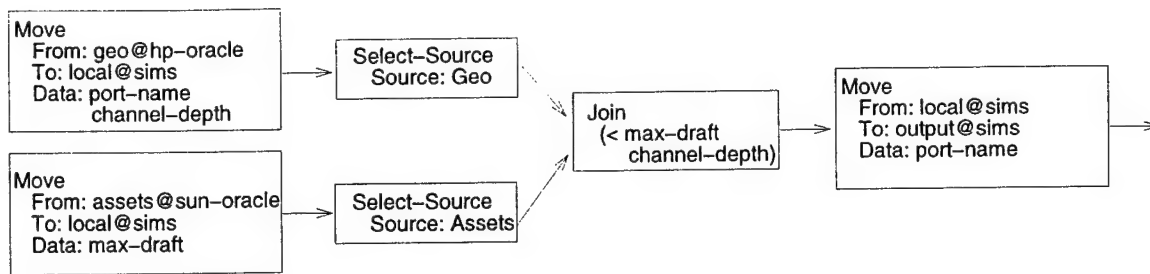
Table 5.3: The join operator

40

Figure 5.1: An information gathering plan

is to find an efficient solution. The first difference makes the problem somewhat easier, while the second difference makes the problem significantly harder since it may require searching a large space of plans.

In order to generate query access plans efficiently, we have carefully constrained the space of possible plans. We wrote the operators such that they generate only the relevant portions of the search space. Some examples of this are: first, the operators only reason about joins in the local system, since joins in the remote systems will be handled by the remote database management system and the planner has no control over how or in what order these are performed. Second, the operators consider only joins across data that are distributed in different information sources. It will generally be less efficient to pull two sets of information from the same information source and perform the join locally rather than in the remote source. Third, since we usually do not have write access to the remote databases, information can only be moved from the remote systems to the local system or directly to the output. However, even with a set of carefully designed operators, the search space may still be very large since the operations can be performed in different orders, and there may be multiple replicated and overlapping sources from which the information can be retrieved.

To further constrain the overall search for an efficient plan, we also employ standard database estimation techniques to write an evaluation function to guide the search. The planner uses the evaluation function in a branch-and-bound search, estimating the cost of each intermediate plan and selecting the plan with the lowest overall execution cost. The cost of each operation is estimated by maintaining information about the size of each relation and the number of different possible values for each attribute of a relation. Assuming a uniform distribution of the data, we then estimate the amount of intermediate data that will be retrieved and manipulated, which is usually the dominant cost in handling multidatabase queries. Using the estimated cost of each operation, we can then compute an estimate for the overall cost of a plan, taking into account the parallelism of some of the actions. The evaluation function allows the planner to compare different partial plans; those plans that are more expensive than the plan eventually selected will never be expanded further.

The final plan generated for the example query in Table 5.1 is shown in Figure 5.1. This plan shows where the information is retrieved from and how the information is manipulated to produce the requested data. The system works backward from the goal to produce a plan to retrieve the data. In this particular plan the final move operator is used to achieve the original goal of sending the requested data to the output; it also generates the subgoal of getting the data into the local system. Next, the system considers how to get the data into the local system and since the information is not available in any single information source, it selects the join operator, which decomposes the original goal into two simpler information goals. Each of these simpler goals is then achieved by using the select-source operator to select a relevant source for each of the requests and translate the requests into subgoals that use the terminology of the selected information source. These goals are in turn achieved by moving the information from the remote information sources into the local system. When this plan is executed, all of the information is brought into the local SIMS mediator, where the draft of the ship can be compared against the depth of the seaports. Once the final set of data has been generated, it is sent to the output.

The approach of searching the space of plans to find the best one is similar to what is done in other systems for producing query plans for relational databases [65]. These systems typically generate the space of query access plans, constraining the space of plans with appropriate domain-specific heuristics, and then evaluate the plans and select the best one. An important difference from traditional query planning systems is that in those systems the source from which the information is to be retrieved is fixed, whereas part of the planning process described here includes the selection of an appropriate information source. While this makes the problem harder, it also provides a much more flexible approach to integrating distributed and heterogeneous sources of information.

So far we have described the approach to generating query plans for information gathering in a distributed and heterogeneous environment. In addition to generating a plan, the system must also execute it. However, unlike traditional database environments, there are a number of problems and issues that arise when dealing with distributed and autonomous information sources. Information sources may be unavailable, queries may fail, new information requests may arise that compete for resources with the currently executing plan, and additional information may be required to select an appropriate plan or formulate an efficient query. In the remainder of this chapter we will describe how planning and execution are tightly integrated and how this integration is used to address the issues that arise during execution.

3   Integrating Planning and Execution

Planning and execution are tightly integrated by considering execution as an integral part of the planning process. This is done by treating the execution of each individual action as a necessary step in completing a plan. The goal of the planner becomes

producing a complete and executed plan rather than just producing a complete plan. Just as achieving all of the preconditions of a plan is required for a complete plan, executing each of the actions is also part of the final result.

Sage keeps track of the current status of every action in the plan by marking them as either *unexecuted, executing, completed,* or *failed.* This is similar to how execution was integrated into IPEM [5]. The underlying planner, UCPOP, maintains a list of *flaws,* which is an agenda of things that need to be done to complete a particular plan. These flaws include *open conditions,* which are subgoals that have not yet been achieved, and *threats,* which are potential interactions between operators that must be resolved by adding ordering or binding constraints. We integrated execution in Sage by adding two new types of flaws: an *unexecuted* action flaw and an *executing* action flaw. Whenever a new operator is added to a plan, the corresponding flaw indicating that the action is unexecuted is also added to the agenda. The *executing* flaw is used to handle the fact that actions are not instantaneous and in some cases may take considerable time. A plan is not complete until all *unexecuted* and *executing* flaws have been removed.

The choice of when to execute an action in a plan is important, since undoing an executed action may be costly or impossible. An action cannot be executed until every precondition of the action has been both planned and achieved by executing the preceding actions. Even after an action is executable, Sage delays execution as long as possible to avoid committing to a partially constructed plan prematurely. Once an action has been executed, it is viewed as a commitment to the plan in which the action occurs – the planner cannot consider any plans that are not refinements of the plan being executed. The idea is that the planner should find the best complete plan before any action is executed. Then once execution is initiated, it resolves any failed subplans or new goals before executing the next action. This means that the planner will never execute an action until the corresponding plan is selected as the best available.

Since executing an action may take considerable time, the planner cannot simply execute an action and wait for the results. Instead, Sage creates a subprocess that executes the action and notifies the planner once it has completed. In order to keep track of the actions currently being executed, the corresponding *unexecuted* flaw is removed from the agenda and the *executing* flaw is added. At any one time there may be a number of actions that are all executing simultaneously. On each cycle of the planner, the system checks if any executing actions have completed. Once an action is completed, the *executing* flaw is removed from the agenda. If it completes successfully, the action is marked as completed. Other actions that depend on this action may now be executable if all of the other preceding actions have also been executed. If an action fails, the failed portion of the plan is removed and then replanned, as described in the next section.
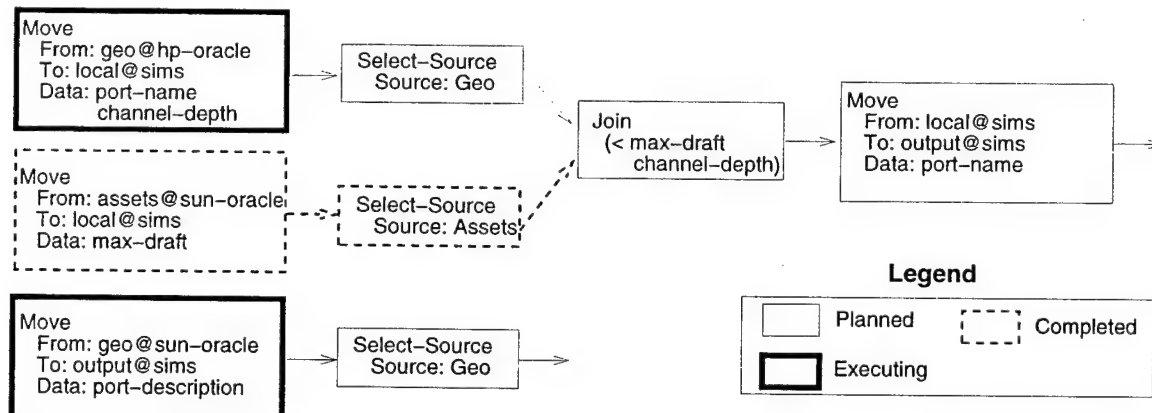
Figure 5.2: Planning for new goals

Sage's top-level algorithm for tightly integrating planning and execution is summarized in Table 5.4. The planner starts with an initial plan, where the goals are the open conditions. Initially, the set of *current plans* contains only this initial plan. It repeats the algorithm until it produces a plan in which every action has been executed. The planner considers only refinements of the *current plans.* Whenever an action is executed, an action terminates, or a new goal is added, the set of *current plans* is replaced by a new set containing only this new plan. The first two conditions in this algorithm ensure that the planner finds a plan with no open conditions or threats before it commits to a plan and initiates any actions.

This algorithm supports simultaneous planning and execution. Before the system initiates execution of any action, it constructs an initially complete plan. However, once execution starts, an action could fail, a new goal could arise, or the system may require additional information (sensing) to continue planning. In any of these cases, once the new open condition has been added to the list of flaws, the system can augment the executing plan to achieve these conditions while it continues executing any actions that have already been initiated. In the next section we describe these capabilities in more detail.

4 Advantages of Integrating Planning and Execution

Integrating the planning and execution allows the system to plan for new goals as they arrive, replan failed actions, and exploit sensing operations, all while the system is executing other actions in a plan.

Remove a plan from the set of *current plans* and apply the first applicable condition:

- If there are any threats, resolve them by adding additional constraints to the plan. Add the possible refinements to the *current plans*.

- If there are any open conditions, add additional actions or ordering links to achieve them. Add the possible refinements to the *current plans*. (As described in the next section, open conditions that contain runtime variables for sensing will be postponed.)

- If any executing actions have completed:

  - If the action completed successfully, record the results and update the plan. If the plan is complete, return the results. Otherwise, replace the *current plans* with this new plan.

  - If the action failed, remove the failed portion of the plan, update the model to avoid generating the same plan again, and replace the *current plans* with this new plan.

- If there are any new goals to solve, add them to the open conditions and replace the *current plans* with this new plan.

- If any unexecuted actions are now executable, create a process to execute them and replace the *current plans* with this new plan.

Table 5.4: Algorithm for planning and execution

4.1 Planning for New Goals

Interleaving planning and execution allows the system to handle new goals while the system is in the midst of executing a plan that achieves some other goals. This is important, since execution may require substantial amounts of time and it may be impractical and inefficient to wait for one task to complete before starting the next task. In addition, it may not be possible to treat the new goal as an independent task since it may compete with the executing plan for the same resources. The handling of new goals is captured in the algorithm described in Table 5.4. When a new goal arises, the system adds this goal to the currently executing plan and then refines that plan to solve the goal.

Consider an example where a new goal is given to the system while it is executing the plan in Figure 5.1. Assume that the system has already executed some of the actions and is in the midst of executing others, as shown in Figure 5.2. When a new goal arises to retrieve the description of the Long Beach seaport, the planner notices the pending goal on the next cycle and then searches for appropriate additions to the currently executing plan to solve this goal. While the system is generating this plan, the action in progress (shown by the action in the box with thick lines) continues to execute, since actions are run as separate processes.

The resulting plan is shown in Figure 5.2. The advantage of planning this new goal in the context of the existing plan is that shared work can be exploited and any potential resource conflicts are considered in the planning process. In this case, the goal requires access to the geo database, which is already in use by the other executing query. As a result, the system uses the geo database running on the sun-oracle server, since the other action that required this resource has already completed. The separate top-level goals are treated as independent goals, so if a subplan fails it will not cause unrelated goals to fail. In addition, as soon as any top-level goal is complete, the results are sent to the calling process. This allows the planner to run continuously and return results as soon as they are obtained rather than waiting for a plan to complete.

4.2 Replanning Failed Actions

Integrating planning and execution allows the system to gracefully handle action failures and replanning. Since the planner may have expended considerable effort in executing a plan so far, we want to avoid throwing out the entire plan and starting from scratch when an action fails. Instead, the planner should replan the failed portion of the plan, while maintaining as much of the executing plan as possible. This is currently supported by requiring the designer of a domain to define a set of domain-specific failure handlers. When a failure occurs, the failure handler is called with the action that failed and the type of failure, and the failure handler is expected to remove the failed portion of the plan and update the model to avoid the same failure when the failed actions are replanned. This replanning can be performed while other unaffected actions continue to execute. A more
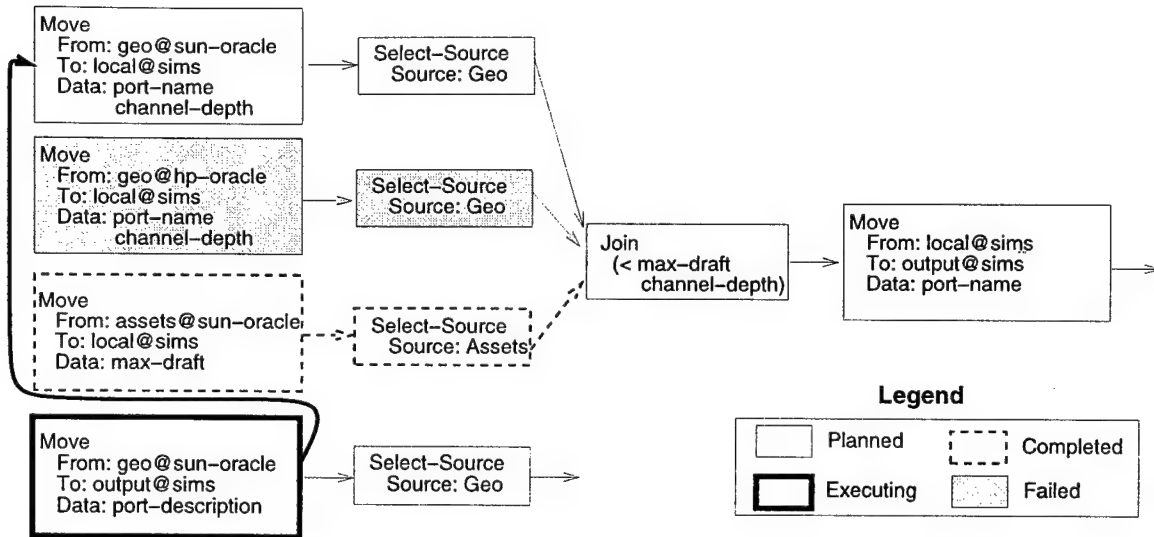
43

Figure 5.3: Replanning a failed plan

complete replanning capability could be incorporated by using the approach developed in the Systematic Plan Adaptor (SPA) [32], which systematically searches the space of plan modifications.

In the information gathering domain, the ability to replan upon failure can be exploited to handle query failures by redirecting a query to a different information source. An execution failure may occur because a database or network is down. In this case the failure handler would remove the actions for retrieving the data from a specific information source and would mark the information source as unavailable to avoid generating the same plan. The planner would then attempt to replan the query; if another information source is available it would generate an alternative plan.

An example of a failed action that can be replanned is shown in Figure 5.3. The actions in the shaded boxes are the failed actions and the actions above the failed ones are the replanned actions. Since the replanned move action requires the same resource as the action currently being executed, an ordering constraint is added between these two actions. This constraint prevents the replanned move action from being executed until this other action completes.

4.3 Sensing to Plan
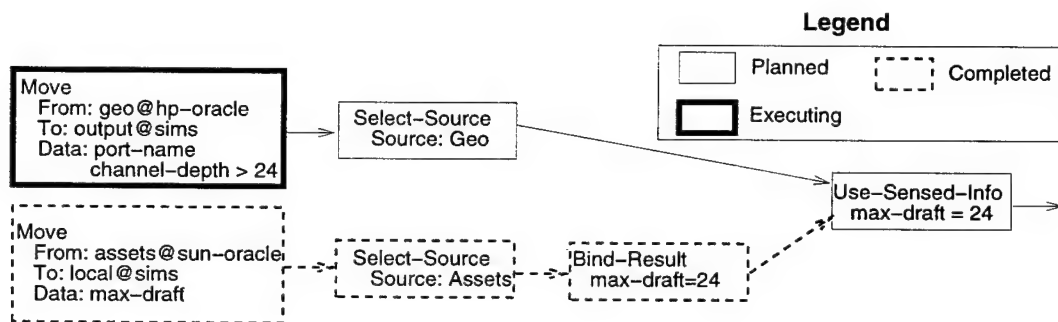
Figure 5.4: Exploiting sensing actions

Integrating planning and execution allows the system to interleave sensing actions with the planning. Earlier work on sensing in planning [5, 21] proposed the idea of incorporating run-time variables in the planner to allow the planner to reason about the sensed information. Run-time variables appear in the effects of operators and essentially serve as place holders for the value or values returned by the action at the point it is executed. These variables are useful because the result can be incorporated and used in other parts of the plan. An issue that arises in the use of run-time variables is that until desired information is available, the planning may have to be postponed or a plan with all possible contingencies will have to be produced in order to deal with the possible returned values. Sage supports run-time variables and delays working on any open condition that involves such a variable. However, unlike previous planners, Sage can begin execution of other actions while it is waiting for the sensed information and then continue planning while these actions continue to execute.

For information gathering, there are two important uses of run-time variables. First, the run-time variables can be used to retrieve information from one source and that information is then used to formulate queries to another source. Second, the run-time variables also can be used to retrieve information which is then used in the selection of the most appropriate information sources. We have already implemented the first use, which is described below, and we investigate the second in [42].

The capability for gathering information to use in the formulation of another query can be added to the system by adding two more operators to the domain, shown in Table 5.5. The first operator is simply an action to execute a query in the local system and bind the result to "!result". As in UWL, run-time variables are annotated with an exclamation mark. The only precondition of this operator is that the information is available in the local system and the only effect is that the data is bound to the result. Note that the system will have to generate a subplan and execute it in order to get the information into the local system.

```
(define (operator bind-result)
    :parameters (?query !result)
    :precondition (available local sims ?query)
    :effect (sensed ?query !result))

(define (operator use-sensed-info)
    :parameters (?source ?host ?query
                 ?mod-query ?sub-query ?result)
    :precondition
       (:and (sensed ?sub-query ?result)
             (available ?source ?host ?mod-query)
             (gather-data ?query ?mod-query
                          ?sub-query ?result))
    :effect (available ?source ?host ?query))
```

<div align="center">Table 5.5: Operators for sensing</div>

The second operator, called use-sensed-info, retrieves information and uses it in the formulation of another query. The heart of this operator is the gather-data precondition, which is a function that determines whether a query can be decomposed such that some of the information can be retrieved and incorporated directly into another query. If so, then it decomposes the original query into a modified query and a sub-query, which will get executed first to return an answer. The result will then be inserted directly into the modified query through the run-time binding.

Consider the example query described in the previous sections. Instead of executing two parallel queries, the system can first gather the information on the ship draft and incorporate that information directly into the second query, as shown in Figure 5.4. In this plan the binding for the "max-draft," is incorporated directly into the query against the geo database. While the two queries must then be done sequentially, it will greatly reduce the amount of intermediate data that needs to be retrieved from the second query. Also, there will be no local processing, so the result can be sent directly to the output.

5   Related Work

There are a variety of systems that have tightly integrated planning with some combination of execution, sensing, and replanning. There is work on reactive planning (e.g., [24, 11]), which emphasizes the ability to react to unexpected situations rather than assume that a plan will usually work. This view is appropriate for some domains, such as robot planning, but not in domains such as information gathering where the cost of execution will usually be much higher than the cost of reasoning about actions. In a partial-order planning framework, Ambros-Ingerson [5] developed an integrated planning, execution, and monitoring system called IPEM and introduced the idea of run-time variables for sensing. Olawsky and Gini [55] focused on the tradeoffs and strategies in choosing when to sense and when to plan. Etzioni et al.[21] developed a language for representing incomplete information and Etzioni et al.[20] built an integrated system for planning, execution, and sensing called XII that can represent and reason about locally complete information. We have built on many of the ideas from the earlier work within the partial-order planning paradigm and extended them to support simultaneous planning and execution and build an integrated system for information gathering.

The other aspect to this work is the application of the planner to the problem of information gathering. The XII planner [20], which is used in the Unix Softbot [22], also supports execution and sensing for information gathering. Compared to Sage, the Softbot reasons about the information at a different level of granularity. Instead of representing general actions for manipulating data, each operator corresponds to a Unix command. The advantage of their approach is that it provides finer-grained control and reasoning of the information. The disadvantage is that it would be impractical to efficiently reason about and manipulate large amounts of information. Information gathering is also similar to conventional query processing in databases. These systems generate a query access plan and then execute it [34]. There is no choice of which information source is used and no capability for interleaving the planning and execution, performing sensing operations, replanning due to failures, or handling additional goals.

6 Discussion

This chapter presented a planning system, called Sage, which tightly integrates planning and execution, runs continuously and handles new goals as they arrive, performs sensing actions, and recovers from failures that arise, all while continuing to execute actions already in progress. The contributions of this work are twofold. First, we extended the previous work by tightly integrating these components and adding the capability to execute actions simultaneously with the planning, replanning, and sensing. Second, we demonstrated that the resulting planner can be effectively applied to the problem of information gathering from distributed and heterogeneous information sources.

In this work we started with a real-world planning application and identified the issues that had to be addressed to solve this problem. While there is a significant amount of previous work on planning that we could build on, the emphasis and assumptions in previous work do not closely match the problems that arise in this domain. For example, in terms of generating plans, the interactions between actions do arise, but they are not the dominant problem. Issues that are important in this domain are finding high quality plans, exploiting parallelism in the plans, and planning and executing simultaneously to support planning for new goals, replanning and sensing. In order to put all of this work together and turn it into a practical planning system, the resulting planner makes some simplifying assumptions that may not hold in other domains. However, the basic architecture is quite general and has been demonstrated in a real-world application.

Sage serves as the underlying query planner for the SIMS information mediator [7, 41]. The goal of SIMS is to provide flexible and efficient access to large numbers of information sources. We have implemented the planning, execution, replanning, and sensing as described in this chapter. The current system has been used in the domains of logistics planning and trauma care and provides access to data stored in a variety of systems that are distributed at various sites.

46

Chapter 6

Model Construction with Key Identification

1 Introduction

A conceptual model of a database is a specification of objects, attributes, and their relationships in the database. Such a model is not only important for understanding the given data but also critical for applications such as reverse engineering and information integration. However, obtaining a conceptual model from a legacy database is difficult task. This is because legacy databases often have incomplete schema, noisy data, and missing information about primary keys and foreign keys.

Although model construction with key identification is an important problem, it has not received much attention in the past and its complexity has often been underestimated. There are at least two mis-concepts in this regard. First, because keys and functional dependencies are very closely related, it is often believed that keys could be found as by-products of functional dependencies. This belief has ignored the fact that legacy databases have large sizes and finding all functional dependencies is not only undesirable but also impractical. Second, it is believed that a simple brute force search for all attributes that have unique values would be sufficient. This approach forgot the fact that in a legacy database, you either find too many such "unique attributes" due to the nature of the data, or you find too few such attributes due to the noise in the data.

Furthermore, key identification must be put into context in order to obtain practical solutions. By exploiting the relation between key identification and model construction, we have developed an approach that can identify primary keys and foreign keys with a high accuracy for the construction of a conceptual model. The main idea is to use the constraints between keys, foreign keys, and the model construction to filter out spurious key information and guide the search toward fruitful directions. This constraint-based approach is also efficient and practical. For the three large legacy databases that have been tested, this approach has obtained comprehensive entity-relationship models in several hours where constructing such models based on conventional database tools may require several man-weeks of work.

In the rest of the chapter, Section 2 highlights the related work, Section 3 provides the necessary definitions, Section 4 describes the details of the constraint-based key identification for model construction and an algorithm called MCKI that implements the idea, Section 5 presents the experimental results of applying MCKI to three large legacy databases, and finally Section 6 concludes the chapter with some future work.

2 Related Work

Although there is little work on model construction with key identification, there is a large body of work on model construction with given key information. For example, [CBS94,C95] describes an approach for extracting an extended entity-relationship (EER) model from a relational database that assumes consistent key names, error-free key values, and primary keys. [MM90] provides an algorithm to generate EER schemas from relational schemas that

47

must consist of relation-schemes, key dependencies, and key-based inclusion dependencies. Earlier approaches on schema translation such as [DA88], [BM89] and [SK92] are also related, although these approaches do not consider data at all.

In KDD research community, model construction has received some limited attention. [MK91] applies an ID3-style algorithm to analyze frame-based data structures in the KATE system. [RKK95] proposes a method of analyzing individual relations and combining the results across databases based on primary and foreign keys. [MR96] presents a reverse engineering method for inferring the background domain knowledge from a database by analyzing the dependencies. [MR96] describes model construction approach by searching for unique or near-unique attributes as keys and then grouping relevant attributes to form complex objects. However, no application to real databases has been reported yet.

3   Basic Definitions

Let R be the schema of a relational database, and R_i be a relation schema of R. In this chapter, we assume R satisfies 2NF, that means all attributes are atomic and there is no partial dependency in the keys of any R_i. A relational database can be defined as a tuple (R, F, I), where F is a set of functional dependencies, and I a set of inclusion dependencies, defined as follows:

A *functional dependency* over a relational schema $Ri(A_{i1}, A_{i2}, ..., A_{in})$ is an expression R_i: X - Y, where X and Y are subsets of R_i. If r_i is a relation over R_i, then R_i: $X \rightarrow Y$ holds in r_i if and only if for any tuples t, t' in r_i, if t[X] = t'[X], then t[Y] = t'[Y]. Two particular types of functional dependencies are of interesting: (1) If R_i: $X \rightarrow R_i$, where X is a subset of R_i, then X is a *super key* of R_i; (2) Since R satisfies 2NF, there exists a primary key that uniquely defines a relation r_i of every R_i.

An *inclusion dependency* over R is an expression $R_i[X] \rightarrow R_j[Y]$, where R_i and R_j are relation schema of R, and X and Y are equal-length sequences of attributes of R_i and R_j, respectively. An inclusion dependency specifies a reference integrity constraint of R_i. For an inclusion dependency $R_i[X] \rightarrow R_j[Y]$, if Y is the super key of R_j, this dependency is called *key based*. Moreover, if Y is a primary key of R_j, the dependency is called *primary key based*, and X is called a *foreign key* of R_i.

In this chapter, the model to be constructed is an entity-relationship model. To do so, each relation R_i will be classified as either an *entity relation* (to be converted into entity) or a *relationship relation* (to be converted into a relationship). In general, a relation's classification is based on the collective information about its primary keys and related foreign keys.

4   Constraint-Based Model Construction with Key Identification

The process for model construction with key identification is as follows:

1.   Identify candidate keys for each relation R_i in R;

2.   Identify candidate foreign keys based on the candidate keys found in 1;

3.   Eliminate those candidate keys that are not referred by any of the foreign keys found in 2;

4.   Identify entity relations based on the candidate keys and foreign keys;

5.   Eliminate those foreign keys that do not point to any entity relations;

6.   Classify each relation as either an entity relation or a relationship relation;

7.   Construct an ER model based on the classified relations.

Notice that each step in this process is either gathering information, or eliminating spurious information based on some constraints that imposed by the gathered information. For example, after candidate keys are found in step 1, they are used to constraint the search for foreign keys in step 2. After foreign keys are found in step 2, they are used to eliminate the spurious candidate keys (those that are not referred by any foreign keys) in step 3, and so on. The details of each step are described as follows.

4.1. Identify Candidate Keys

To identify a set of candidate keys for a given relation $R_i(A_{i1}, ..., A_{in})$, we check the values in a subset X of R_i, and calculate the likelihood of X being a key as follows:

$p(X)$ = (the number of unique tuples in X)/(the number of tuples in R_i).

We then choose those X that have value $p(X) > g$, where g is a predetermined threshold. For example, g can be set as such that every relation should have at least one candidate key. To constraint the search, we also limit the length of X to be less than 5. (This number is determined empirically for we believe that the probability of having a compound key having more than 5 attributes is very low.) The result of this step is a set of candidate key for every relation. As one may expect, more than one candidate key may be found for a relation and not all of them are genuine. For example, an attribute with the type of real number is likely to have many unique values, but it is not a key. On the hand, since legacy databases may contain much noisy data, some relations may not have attributes that qualify as keys at all. Such relations will be reported to the user as special cases.

4.2. Identify Candidate Foreign Keys

Given the candidate keys found in the last step, the system is then search for inclusion dependency $R_i[X] \rightarrow R_j[Y]$, where Y is a candidate key for R_j, in order to find candidate foreign key X for R_i. Notice that the search for inclusion dependency is not blind but purposely directed to candidate keys (i.e., Y must be a candidate key). In contrast to other approaches, this constraint has contributed greatly to the efficiency of the entire search process.

4.3. Eliminate Spurious Candidate Keys

In this step, the candidate foreign keys found in the last step are used as constraints to eliminate spurious candidate keys. The idea is as follows. If both X and Y are candidate keys for a relation R_i, and X is referred by some foreign keys while Y is not, then Y will be eliminated from the set of candidate keys for R_i. The justification is that if a candidate key is genuine, then it should be referred to by other relations when such references exist. When more than one candidate keys are referred by foreign keys, then the one that is referred most frequently will be select as the primary key for the relation.

4.4. Identify Entity Relations

At this stage, enough information has been accumulated to allow classification of entity relations. In particular, a relation R_i is classified as an entity relation if its primary key does not contain any foreign keys. We omit the justification but readers can found them in [CBS94].

4.5. Eliminate Spurious Foreign Keys

Once entity relations are identified, they can be used to eliminating spurious foreign keys. The idea is simple, if a candidate foreign key does not point to any of the entity relations, then it does not contribute to the final construction of ER model, and thus can be removed from the set of possible foreign keys. After this step, we have identified sufficient (and possibly necessary) information for properly classifying relationship relations in the next step.

4.6. Classify Relationship Relations and Other Entity Relations

In this step, all the relations that have identified primary key and foreign keys will be classified. The decision whether a relation Ri should be classified as an entity or a relationship is based on the interdependency between its primary key and foreign keys, as follows.

If Ri has only one foreign key, and this foreign key is contained in the primary key, then classify Ri as an entity relation;

If Ri has two foreign keys and they form the primary key of Ri, then classify Ri as a relationship relation. This relationship will link the two entity relations pointed to by the foreign keys;

If Ri has two foreign keys and one is in the primary key and the other is not, then classify Ri as an entity relation. This entity will have two relationships through the foreign keys.

If Ri has n (n>2) foreign keys, and at least one of them is contained in the primary key, then classify Ri as a n-ary relationship relation, with links to the n entity relations pointed to by the foreign keys. To illustrate the idea,

49

consider the following divorce case,

MAN(SSN, Name, Bdate),
WOMAN(SSN, Name, Bdate),
LAWYER(SSN, Name, Bdate),
DIVORCE(ManSSN, WomanSSN, LawyerSSN, Date)

The primary key of DIVORCE is ManSSN (or WomanSSN). It has three foreign keys reference to MAN, WOMAN, and LAWYER respectively. We can convert DIVORCE to a 3-ary relationship among the three entities.

4.7. Construct an ER Model Based on The Classified Relations

With all the relations classified, this step simply constructs a ER model according to the classified relations. All entity relations will be converted into entities, and their names and attributes (except the foreign keys) will also be carried over. All the relationship relations will be converted into relationship with foreign keys are the pointers to the appropriate entities. They will use the same name, and all their attributes will be made as attributes of the relationship.

5 Experimentation

To evaluate this approach, we have implemented the above algorithm in a system called MCKI (Model Construction with Key Identification), and tested it on three real databases: ALPI, CVI and GITI8I, collected from some daily-used logistics applications. All of them have the typical characteristics of legacy database: (1) They contain large amounts of data. ALPI has 67 tables, with maximum 60 attributes and 515,668 rows. CVI has 104 tables, with maximum 38 attributes and 99,264 rows. GITI8I has 48 tables, with maximum 20 attributes and 1,480,445 rows. (2) They miss very important schema information. There is no specification for primary key, foreign key, and domain experts are almost impossible to access. (3) They contain a lot of noisy data. For example, most tables in these databases have spurious candidate keys (set of attributes that have unique values). In other word, these databases are not even in 2NF on the first glance.

MCKI has been tested on all three databases. Due to space limitation in this chapter, we will only report the result in ALPI.

5.1.1.1	Table 1. Examples of Candidate Keys found in ALPI

#Table8:CIF_FAILURE_FACTOR: numRows=808, numColumns=7, keys=1
EI_LIN, PART_NSN : 1.000000
#Table9:COLUMN_ATTRIBUTES: numRows=120, numColumns=4, keys=3
TABLE_NAME, COLUMN_NAME : 0.958333
TABLE_NAME, ATTRIBUTE_NAME : 1.000000
TABLE_NAME, ATTRIBUTE_VALUE : 1.000000
#Table10:DATABASE_HISTORY: numRows=36, numColumns=10, keys=5
TABLENAME : 0.916667
TABLENAME, USERNAME : 0.916667
TABLENAME, TIMESTAMP : 1.000000
TABLENAME, MOD_TYPE : 0.916667
TABLENAME, DATAFILE : 1.000000
#Table11:DLA_STOCK_REPORT_1: numRows=51784, numColumns=7, keys=1
NSN, RIC : 0.703808
#Table36:LOGAD_TABLES: numRows=120, numColumns=9, keys=2
TABLE_NAME : 0.983471
PRETTY_NAME : 1.000000
#Table41:MP_STOCK_REPORT_1: numRows=64891, numColumns=11, keys=1
NSN, STATION_CODE, : 1.000000
#Table43:NSN_DESCRIPTION: numRows=25716, numColumns=60, keys=2
NIIN, : 1.000000
NSN, : 1.000000
#Table58:TAV_CODES: numRows=752, numColumns=4, keys=5
VALUE_1, : 0.918883
CODE_TYPE, CODE, : 1.000000
CODE_TYPE, VALUE_1, : 0.928191
CODE, VALUE_1, : 1.000000
VALUE_1, VALUE_2, : 0.953457

```
#Table61:TAV_STOCK_REPORT_1: numRows=4926, numColumns=11, keys=1
NSN, UIC_RIC, : 0.944580
#Table62:TAV_STOCK_REPORT_2: numRows=1532, numColumns=11, keys=1
NSN, UIC_RIC, : 0.843342
#Table63:TAV_STOCK_REPORT_3: numRows=513497, numColumns=11, keys=1
NSN, UIC_RIC, : 0.947355
#Table64:TAV_STOCK_REPORT_4: numRows=515668, numColumns=11, keys=1
NSN, UIC_RIC, : 0.962005
```

As we can see in Table 1, many tables in this database has more than one candidate keys, and due to the noisy data, some tables do not have candidate keys at all. Table 2 shows some examples of candidate foreign keys found by MCKI. As we can see there, candidate foreign keys in this database are very well organized. Most of them point to the same candidate key if they refer to the same table. Using these candidate foreign keys, primary keys and spurious candidate keys can easily be identified. For example, in Table 36, although both PRETTY_NAME and TABLE_NAME are qualified as keys, TABLE_NAME is selected as the primary key (it has 8 foreign key references) while PRETTY_NAME is eliminated (it has no foreign key reference), even if it has a higher value of uniqueness than TABLE_NAME (1.0 vs.0.983477).

Table 2. Examples of candidate foreign keys found in ALPI

```
#Table10:DATABASE_HISTORY: numRows=36, numColumns=10, keys=5, foreigns=1
        (TABLENAME): LOGAD_TABLE_SEGMENTS(TABLE_NAME);
        (TABLENAME,USERNAME): No FKEY REFERENCE!
        (TABLENAME,TIMESTAMP): No FKEY REFERENCE!
        (TABLENAME,MOD_TYPE): No FKEY REFERENCE!
        (TABLENAME,DATAFILE): No FKEY REFERENCE!
#Table22:GLAD_META_JOIN_TO: numRows=23, numColumns=4, keys=4, foreigns=1
        (JOIN_NAME): GLAD_META_JOIN_FROM(JOIN_NAME);
        (JOIN_NAME,TABLE_NAME): No FKEY REFERENCE!
        (JOIN_NAME,COLUMN_NAME): No FKEY REFERENCE!
        (TABLE_NAME,COLUMN_NAME): No FKEY REFERENCE!
#Table29:ITEM_GROUP_2: numRows=4, numColumns=1, keys=1, foreigns=1
        (NSN): CIF_FAILURE_FACTOR(EI_NSN);
#Table32:LOGAD_COLUMNS: numRows=1494, numColumns=16, keys=4, foreigns=1
        (TABLE_NAME,COLUMN_NAME): GLAD_META_JOIN_FROM(TABLE_NAME,COLUMN_NAME);
        (TABLE_NAME,PRETTY_NAME): No FKEY REFERENCE!
        (TABLE_NAME,SHORT_NAME): No FKEY REFERENCE!
        (TABLE_NAME,DOCUMENTATION): No FKEY REFERENCE!
#Table36:LOGAD_TABLES: numRows=121, numColumns=9, keys=2, foreigns=8
        (TABLE_NAME): DATABASE_HISTORY(TABLENAME);
                GLAD_META_JOIN_FROM(TABLE_NAME);
                GLAD_META_PRETTY_NAMES(TABLE_NAME);
                LOGAD_CODES(TABLE_NAME);
                LOGAD_COLUMNS(TABLE_NAME);
                LOGAD_INDICES(TABLE_NAME);
                LOGAD_POST_LOAD_PROCESSING(TABLE_NAME);
                LOGAD_TABLE_SEGMENTS(TABLE_NAME);
        (PRETTY_NAME): No FKEY REFERENCE!
#Table43:NSN_DESCRIPTION: numRows=25716, numColumns=60, keys=2, foreigns=3
        (NIIN): No FKEY REFERENCE!
        (NSN): CIF_FAILURE_FACTOR(EI_NSN);
                TAV_STOCK_REPORT_1(NSN);
                TAV_STOCK_REPORT_2(NSN);
#Table51:RIC: numRows=5110, numColumns=9, keys=1, foreigns=1
        (RIC): CIF_FAILURE_FACTOR(SOURCE_OF_SUPPLY);
#Table59:TAV_NSN_TEMP: numRows=3327, numColumns=1, keys=1, foreigns=5
        (NSN): CIF_FAILURE_FACTOR(EI_NSN);
                TAV_STOCK_REPORT_1(NSN);
                TAV_STOCK_REPORT_2(NSN);
                TAV_STOCK_REPORT_3(NSN);
                TAV_STOCK_REPORT_4(NSN);
#Table66:WORLD_DESCRIPTION: numRows=1, numColumns=2, keys=2, foreigns=2
        (WORLD_ID): DATABASE_HISTORY(WORLD_ID);
                DLA_STOCK_REPORT_1(WORLD_ID);
        (DESCRIPTION): No FKEY REFERENCE!
```

With the primary keys and foreign keys found, MCKI constructs an ER model for the ALPI database. The model contains 67 entities and 25 relationships. The entire process takes about 3 hours.

6 Conclusion

This chapter presents a method for model construction with key identification in legacy databases. Compared to most existing approach, this method is data-driven and constraint-based. It does not assume that the primary keys and the foreign keys are given. Instead, the system discovers this information by analyzing the data. One innovation of this approach is that the search for critical information is not brute force, but exploiting constraints that are imposed on the analyzed results. The method has been tested on some very large legacy database and the results are promising.

Bibliography

[1] Sibel Adali, K.S. Candan, Yannis Papakonstantinou, and V.S. Subrahmanian. Query caching and optimization in distributed mediator systems. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 137–148, Montreal, Canada, 1996.

[2] Rafi Ahmed, Philippe De Smedt, Weimin Du, William Kent, Mohammad A. Ketabchi, Witold A. Litwin, Abbas Rafii, and Ming Chien Shan. The Pegasus heterogenous multidatabase system. *IEEE Computer*, pages 19–27, 1991.

[3] José Luis Ambite and Craig A. Knoblock. Planning by rewriting: Efficiently generating high-quality plans. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence*, Providence, RI, 1997.

[4] Jose Luis Ambite and Craig A. Knoblock. Flexible and scalable query planning in distributed and heterogeneous environments. In *Proceedings of the Fourth International Conference on Artificial Intelligence Planning Systems*, Pittsburgh, PA, 1998.

[5] Jose Ambros-Ingerson. *IPEM: Integrated Planning, Execution, and Monitoring*. PhD thesis, Department of Computer Science, University of Essex, 1987.

[6] Jose Ambros-Ingerson and Sam Steel. Integrating planning, execution, and monitoring. In *Proceedings of the Seventh National Conference on Artificial Intelligence*, pages 83–88, Saint Paul, Minnesota, 1988.

[7] Yigal Arens, Chin Y. Chee, Chun-Nan Hsu, and Craig A. Knoblock. Retrieving and integrating data from multiple information sources. *International Journal on Intelligent and Cooperative Information Systems*, 2(2):127–158, 1993.

[8] Yigal Arens and Craig A. Knoblock. Planning and reformulating queries for semantically-modeled multidabase systems. In *Proceedings of the First International Conference on Information and Knowledge Management*, pages 92–101, Baltimore, MD, 1992.

[9] Yigal Arens, Craig A. Knoblock, and Wei-Min Shen. Query reformulation for dynamic information integration. *Journal of Intelligent Information Systems, Special Issue on Intelligent Information Integration*, 6(2/3):99–130, 1996.

[10] Anthony Barrett, Keith Golden, Scott Penberthy, and Daniel Weld. UCPOP user's manual Version 2.0. Technical Report 93-09-06, Department of Computer Science and Engineering, University of Washington, 1993.

[11] Michael Beetz and Drew McDermott. Declarative goals in reactive plans. In *Artificial Intelligence Planning Systems: Proceedings of the First International Conference (AIPS92)*, pages 3–12, College Park, MD, 1992.

[12] R.J. Brachman and J.G. Schmolze. An overview of the KL-ONE knowledge representation system. *Cognitive Science*, 9(2):171–216, 1985.

[13] Saša Buvač. Quantificational logic of context. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence*, 1996.

[14] Christine Collet, Michael N. Huhns, and Wei-Min Shen. Resource integration using a large knowledge base in Carnot. *IEEE Computer*, pages 55–62, December 1991.

[15] Ken Currie and Austin Tate. O-plan: The open planning architecture. *Artificial Intelligence*, 52(1):49–86, 1991.

[16] Tom Dean and Mark Boddy. An analysis of time-dependent planning. In *Proceedings of the Seventh National Conference on Artificial Intelligence*, pages 49–54, Saint Paul, Minnesota, 1988.

[17] Oliver M. Duschka and Michael R. Genesereth. Infomaster - an information integration tool. In *Proceedings of the International Workshop on Intelligent Information Integration*, Freiburg, Germany, September 1997.

[18] Kutluhan Erol, Dana Nau, and V. S. Subrahmanian. Decidability and undecidability results for domain-independent planning. *Artificial Intelligence*, 76(1-2):75–88, 1995.

[19] Oren Etzioni. Acquiring search-control knowledge via static analysis. *Artificial Intelligence*, 62(2):255–302, 1993.

[20] Oren Etzioni, Keith Golden, and Dan Weld. Tractable closed-world reasoning with updates. In *Fourth International Conference on Principles of Knowledge Representation and Reasoning*, Bonn, Germany, 1994.

[21] Oren Etzioni, Steve Hanks, Daniel Weld, Denise Draper, Neal Lesh, and Mike Williamson. An approach to planning with incomplete information. In *Proceedings of the Third International Conference on Principles of Knowledge Representation and Reasoning*, pages 115–125, Cambridge, MA, 1992.

[22] Oren Etzioni and Daniel S. Weld. A softbot-based interface to the Internet. *Communications of the ACM*, 37(7), 1994.

[23] Adam Farquhar, Angela Dappert, Richard Fikes, and Wanda Pratt. Integrating information sources using context logic. In *AAAI Spring Symposium on Information Gathering from Distributed Heterogeneous Environments*, 1995.

[24] R. James Firby. An investigation into reactive planning in complex domains. In *Proceedings of the Sixth National Conference on Artificial Intelligence*, pages 202–206, Seattle, WA, 1987.

[25] Charles L. Forgy. Rete: A fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligence*, 19:17–37, 1982.

[26] David E. Foulser, Ming Li, and Qiang Yang. Theory and algorithms for plan merging. *Artificial Intelligence*, 57(2–3):143–182, 1992.

[27] Koichi Furukawa. A deductive question answering system on relational data bases. In *Proceedings of the Fifth International Joint Conference on Artificial Intelligence*, pages 59–66, Cambridge, MA, 1977.

[28] Geotz Graefe and David J. DeWitt. The EXODUS optimizer generator. *Proceedings of the 1987 ACM SIGMOD International Conference on Management of Data*, 16(3):160–172, 1987.

[29] R.V. Guha. *Contexts: A formalization and some applications.* PhD thesis, Stanford University, 1991.

[30] Laura M. Haas, Donald Kossmann, Edward L. Wimmers, and Jun Yang. Optimizing queries across diverse data sources. In *Proceedings of the Twenty-third International Conference on Very Large Data Bases*, Athens, Greece, 1997.

[31] Joachim Hammer, Hector Garcia-Molina, Kelly Ireland, Yannis Papakonstantinou, Jeffrey Ullman, and Jennifer Widom. Information translation, mediation, and mosaic-based browsing in the tsimmis system. In *Proceedings of the ACM SIGMOD Internation Conference on Management of Data*, San Jose, CA, 1995.

[32] Steven Hanks and Daniel S. Weld. The systematic plan adaptor: A formal foundation for case-based planning. Technical Report 92-09-04, Department of Computer Science and Engineering, University of Washington, Seattle, WA, 1992.

[33] Steven Hanks and Daniel S. Weld. A domain-independent algorithm for plan adaptation. *Journal of Artificicial Intelligence Research*, 2:319–360, 1995.

[34] Matthias Jarke and Jurgen Koch. Query optimization in database systems. *ACM Computing Surveys*, 16(2):111–152, 1984.

[35] R.J. Bayardo Jr., W. Bohrer, R. Brice, A. Cichocki, J. Fowler, A. Helal, V. Kashyap, T. Ksiezyk, G. Martin, M. Nodine, M. Rashid, M. Rusinkiewicz, R. Shea, C. Unnikrishnan, A. Unruh, and D. Woelk. Infosleuth: Agent-based semantic integration of information in open and dynamic environments. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Tucson, AZ, 1997.

[36] Henry Kautz and Bart Selman. Pushing the envelope: Planning, propositional logic, and stochastic search. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence*, Portland, OR, 1996.

[37] Craig A. Knoblock. Automatically generating abstractions for planning. *Artificial Intelligence*, 68(2), 1994.

[38] Craig A. Knoblock. Generating parallel execution plans with a partial-order planner. In *Proceedings of the Second International Conference on Artificial Intelligence Planning Systems*, Chicago, IL, 1994.

[39] Craig A. Knoblock. Planning, executing, sensing, and replanning for information gathering. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence*, Montreal, Canada, 1995.

[40] Craig A. Knoblock. Building a planner for information gathering: A report from the trenches. In *Proceedings of the Third International Conference on Artificial Intelligence Planning Systems*, Edinburgh, Scotland, 1996.

[41] Craig A. Knoblock, Yigal Arens, and Chun-Nan Hsu. Cooperating agents for information retrieval. In *Proceedings of the Second International Conference on Cooperative Information Systems*, Toronto, Canada, 1994.

[42] Craig A. Knoblock and Alon Levy. Exploiting run-time information for efficient processing of queries. In *Working Notes of the AAAI Spring Symposium on Information Gathering in Heterogeneous, Distributed Environments*, Palo Alto, CA, 1995.

[43] Terry Landers and Ronni L. Rosenberg. An overview of Multibase. In H.J. Schneider, editor, *Distributed Data Bases*. North-Holland, 1982.

[44] Alon Y. Levy, Anand Rajaraman, and Joann J. Ordille. Query-answering algorithms for information agents. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence*, Portland, OR, 1996.

[45] Alon Y. Levy, Anand Rajaraman, and Joann J. Ordille. Querying heterogeneous information sources using source descriptions. In *Proceedings of the 22nd VLDB Conference, Bombay, India*, 1996.

[46] Alon Y. Levy and Marie-Christine Rousset. Carin: A representation language integrating rules and description logics. In *Proceedings of the European Conference on Artificial Intelligence*, Budapest, Hungary, 1996.

[47] Alon Y. Levy and Marie-Christine Rousset. The limits on combining recursive horn rules and description logics. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence*, Portland, OR, 1996.

[48] Alon Y. Levy, Divesh Srivastava, and Thomas Kirk. Data model and query evaluation in global information systems. *Journal of Intelligent Information Systems, Special Issue on Networked Information Discovery and Retrieval*, 5(2), 1995.

[49] Robert MacGregor. A deductive pattern matcher. In *Proceedings of the Seventh National Conference on Artificial Intelligence*, Saint Paul, Minnesota, 1988.

[50] Robert MacGregor. The evolving technology of classification-based knowledge representation systems. In John Sowa, editor, *Principles of Semantic Networks: Explorations in the Representation of Knowledge*. Morgan Kaufmann, 1990.

[51] Steven Minton. *Learning Search Control Knowledge: An Explanation-Based Approach*. Kluwer, Boston, MA, 1988.

[52] Steven Minton. Minimizing conflicts: A heuristic repair method for constraint-satisfaction and scheduling problems. *Artificial Intelligence*, 58(1-3):161–205, 1992.

[53] Steven Minton, Craig A. Knoblock, Daniel R. Kuokka, Yolanda Gil, Robert L. Joseph, and Jaime G. Carbonell. PRODIGY 2.0: The manual and tutorial. Technical Report CMU-CS-89-146, School of Computer Science, Carnegie Mellon University, 1989.

[54] Dana S. Nau, Satyandra K. Gupta, and William C. Regli. AI planning versus manufacturing-operation planning: A case study. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence*, Montreal, Canada, 1995.

[55] Duane Olawsky and Maria Gini. Deferred planning and sensor use. In *Proceedings of the Workshop on Innovative Approaches to Planning, Scheduling and Control*, pages 166–174, San Diego, CA, 1990.

[56] Christos H. Papadimitriou and Kenneth Steiglitz. *Combinatorial Optimization: Algorithms and Complexity*. Prentice Hall, Englewood Cliffs, NJ, 1982.

[57] Juergen Paulokat and Stefan Wess. Planning for machining workpieces with a partial-order, nonlinear planner. In *Working notes of the AAAI Fall Symposium on Planning and Learning: On to Real Applications*, November 1994.

[58] J. Scott Penberthy and Daniel S. Weld. UCPOP: A sound, complete, partial order planner for ADL. In *Third International Conference on Principles of Knowledge Representation and Reasoning*, pages 189–197, Cambridge, MA, 1992.

[59] M. Alicia Pérez. Representing and learning quality-improving search control knowledge. In *Proceedings of the Thirteenth International Conference on Machine Learning*, Bari, Italy, 1996.

[60] Daniel Ratner and Ira Pohl. Joint and LPA*: Combination of approximation and search. In *Proceedings of the Fifth National Conference on Artificial Intelligence*, Philadelphia, PA, 1986.

[61] M.P. Reddy, B.E. Prasad, and P.G. Reddy. Query processing in heterogeneous distributed database management systems. In Amar Gupta, editor, *Integration of Information Systems: Bridging Heterogeneous Databases*, pages 264–277. IEEE Press, NY, 1989.

[62] Mary Tork Roth and Peter Schwarz. Don't scrap it, wrap it! A wrapper architecture for legacy data sources. In *Proceedings of the Twenty-third International Conference on Very Large Data Bases*, Athens, Greece, 1997.

[63] Earl D. Sacerdoti. Language access to distributed data with error recovery. In *Proceedings of the Fifth International Joint Conference on Artificial Intelligence*, pages 196–202, Cambridge, MA, 1977.

[64] Andy Schurr. *Programmed Graph Replacement Systems*. World Scientific, 1996.

[65] P. Griffiths Selinger, M.M. Astrahan, D.D. Chamberlin, R.A. Lorie, and T.G. Price. Access path selection in a relational database management system. In *Artificial Intelligence and Databases*, pages 511–522. Morgan Kaufmann, Los Altos, CA, 1988.

[66] Shashi Shekhar and Soumitra Dutta. Minimizing response times in real time planning and search. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, Detroit, MI, 1989.

[67] Jeffrey D. Ullman. *Principles of Database and Knowledge-Base Systems*, volume 2. Computer Science Press, Rockville, Maryland, 1989.

[68] Jeffrey D. Ullman. Information integration using logical views. In *Proceedings of the Sixth International Conference on Database Theory*, Delphi, Greece, January 1997.

[69] Manuela Veloso. *Planning and Learning by Analogical Reasoning*. Springer Verlag, December 1994.

[70] Manuela M. Veloso. Nonlinear problem solving using intelligent casual-commitment. Technical Report CMU-CS-89-210, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, 1989.

[71] Gio Wiederhold. Mediators in the architecture of future information systems. *IEEE Computer*, March 1992.

[72] David E. Wilkins. Domain-independent planning: Representation and plan generation. *Artificial Intelligence*, 22(3):269–301, 1984.

[73] David E. Wilkins. *Practical Planning: Extending the Classical AI Planning Paradigm*. Morgan Kaufmann, San Mateo, CA, 1988.

[74] Qiang Yang, Dana S. Nau, and James Hendler. Merging separately generated plans with restricted interactions. *Computational Intelligence*, 8(4), 1992.

[75] C.T. Yu and C.C. Chang. Distributed query processing. *ACM Computing Surveys*, 16(4):399–433, 1984.

[76] Monte Zweben, Brian Daun, and Michael Deale. Scheduling and rescheduling with iterative repair. In *Intelligent Scheduling*, pages 241–255. Morgan Kaufman, San Mateo, CA, 1994.

55

Appendix

The SIMS Manual
Version 2.0

The SIMS Manual
Version 2.0*

José-Luis Ambite
Yigal Arens
Naveen Ashish
Craig A. Knoblock
Steven Minton
Jay Modi
Maria Muslea
Andrew Philpot
Wei-Min Shen
Sheila Tejada
Weixiong Zhang
Information Sciences Institute and Department of Computer Science
University of Southern California
4676 Admiralty Way,
Marina del Rey, CA 90292, U.S.A.

December 22, 1997

Abstract

SIMS provides intelligent access to heterogeneous, distributed information sources, while insulating human users and application programs from the need to be aware of the location of the sources, their query languages, organization, size, etc.

This manual explains how to bring up a SIMS information server in a new application domain. After providing a short overview of relevant features of the SIMS system, it describes the modeling and programming work that has to be performed to support the extension of SIMS to a given collection of information sources in the domain. To aid a user inexperienced with the technological infrastructure underlying SIMS, the manual contains examples structured as a tutorial that can be followed to actually produce a working SIMS system.

Contents

1 Introduction

The overall goal of the SIMS project is to provide integrated access to information distributed over multiple, heterogeneous sources: databases, knowledge bases, flat files, Web pages, programs, etc. In providing such access, SIMS tries to insulate human users and application programs from the need to be aware of the location of sources and distribution of queried data over them, individual source query languages, their organization, data model, size, and so forth. The processing of user requests should be robust, capable of recovery from execution-time failures and able to handle and/or report inconsistency and incompleteness of data sources. At the same time SIMS has the goal of making the process of incorporating new sources as simple and automatic as possible.

The SIMS approach to this integration problem has been based largely on research in Artificial Intelligence; primarily in the areas of knowledge representation, planning, and machine learning. A model of the application domain is created, using a knowledge representation system to establish a fixed vocabulary for describing objects in the domain, their attributes and relationships among them. Using this vocabulary, a description is created for each information source. Each description indicates the data-model used by the source, the query language, network location, size estimates, etc., and describes the contents of its fields in relation to the domain model. SIMS' descriptions of different information sources are independent of each other, greatly easing the process of extending the system. Some of the modeling is aided by source analysis software developed as part of the SIMS effort.

Queries to SIMS are written in a high-level language (Loom or a subset of SQL) using the terminology of the domain model — independent of the specifics of the information sources. Queries need not contain information indicating which sources are relevant to their execution or where they are located. Queries do not need to state how information present in different sources should be joined or otherwise combined or manipulated.

SIMS uses a planner to determine how to identify and combine the data necessary to process a query. In a pre-processing stage, all data sources possibly relevant to the query are identified. The planner then selects a set of sources that contain the queried information and generates an initial plan for the query. This plan is repeatedly refined and optimized until it meets given performance criteria. The plan itself includes, naturally, sub-queries to appropriate information sources, specification of locations for processing intermediate data, and parallel branches when appropriate. The SIMS system then executes the plan. The plan's execution is monitored and replanning is initiated if its performance meets with difficulties such as unexpectedly unavailable sources. It is also possible for the plan to include explicit replanning steps, after reaching a state where more is known about the circumstances of plan execution.

Changes to information sources are handled by changing source descriptions only. The changes will automatically be considered by the SIMS planner in producing future plans that utilize information from the modified sources. This greatly facilitates extensibility.

The rest of this section presents an overview of SIMS and its architecture. In Section 2 we show the format of the queries that a user would input to SIMS and the output that should be expected. Then we consider in more detail the specification of the domain model, in Section 3, and how information sources are described to the system, in Section 4. Section 5 gives a brief introduction on how to construct a wrapper for a new information source and how to communicate with the wrapper. Section 6 explains how to run SIMS both through its graphical user interface and its functional interface. Section 7 describes how to test and debug a new SIMS application. Section 8 presents the installation and system requirements. Finally, in Section 9 we show the code that would implement the example that is discussed throughout the manual. Section 10 contains a reading list of relevant papers.

1.1 Architecture and Background

A visual representation of the components of SIMS is provided in Figure 1.

SIMS addresses the problems that arise when one tries to provide a user familiar only with the general domain with access to a system composed of numerous separate data- and knowledge-bases.

Specifically, SIMS does the following:

- **Modeling:** It provides a consistent way of describing information sources to the system, so that data
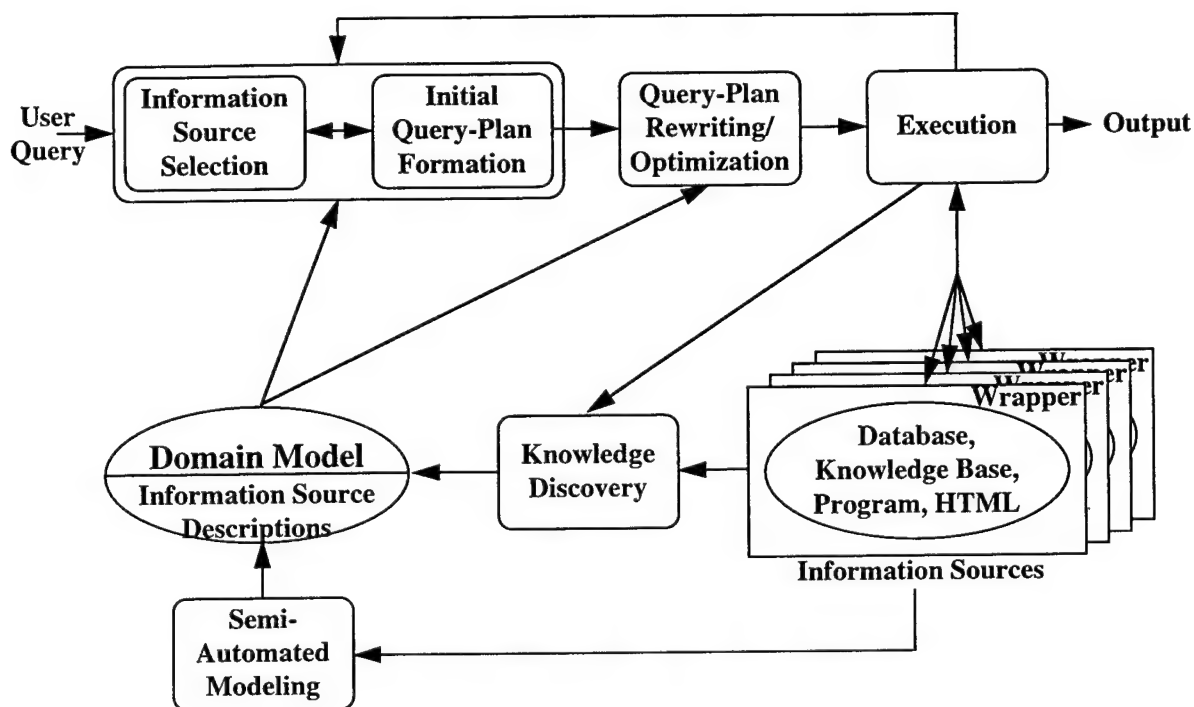
Figure 1: SIMS Overview Diagram.

in them is accessible to it.

- **Information Source Selection:** Given a query, it

  - Determines which classes of information will be relevant to answering the query.
  - Quickly, using some information generated during an earlier preprocessing stage, generates a list of all combinations of sources that contain all information required for a query.

- **Initial Query-Plan Formation:** It creates an initial plan, a sequence of subqueries and other forms of data-manipulation that when executed will yield the desired information. This initial plan does not necessarily satisfy any optimization requirements.

- **Query-Plan Rewriting/Optimization:** By successively applying rewriting rules that preserve the correctness of the plan, it gradually improves the plans efficiency. This process continues until no further rewriting is possibly, or until the allotted time runs out.

- **Semi-Automated Modeling:** By querying databases and other information sources and analyzing the returned information, it discovers semantic rules characterizing their contents. This learned knowledge is used to help create SIMS' information source descriptions.

- **Execution:** It executes the reformulated query plan; establishing network connections with the appropriate information sources, transmitting queries to them and obtaining the results for further processing. During the execution process SIMS may detect that certain information sources are not available, or respond erroneously. In such cases, the relevant portion of the query plan will be replanned. In addition, certain plans will contain steps that require replanning some plan branch some time into the execution phase, after more information is known.

Each information source is accessed through a *wrapper*, a module that can translate from a description of a set of data in SIMS' internal representation language (Loom) into a query for that data that is then

submitted to the source. The wrapper also handles communication with the information source and takes the data returned by it and sends it on to SIMS in the form SIMS expects.

1.2 Information Sources Supported

In order for SIMS to support an information source it must have a description of the source, and there must exist a wrapper for that type of source. While each information source needs to be described individually, only one wrapper is required for any type of information source.

Wrappers for Loom knowledge bases and MUMPS-based network databases have been written for SIMS. In addition, through an "ODBC wrapper" SIMS uses ODBC to interact with all ODBC-enabled databases. This includes Oracle, Sybase, Informix, Ingres, and many others. To add a new database of any of these types requires, therefore, only to create an information source description for it. In order to add an information source of a new type one would have to obtain, or write, a new wrapper for it as well. We are currently working on wrappers for certain object oriented databases. We also have an ongoing associated effort (Ariadne) that includes work on semi-automatic generation of wrappers for HTML pages.

2   The SIMS Query Language

Currently, SIMS only supports commands for retrieving data. Specifically, SIMS takes a retrieval query as input and returns the data satisfying the constraints specified in the query. The output format of SIMS is a tuple of constant(s) or a list of tuples of constant(s). A retrieval query can be expressed in a LOOM syntax or a SQL syntax. The following two sections discuss these two languages in detail.

2.1   LOOM Syntax

Loom serves as the knowledge representation system that SIMS uses to describe the domain model and the contents of the information sources. In addition, Loom is used to define a knowledge base that itself serves as an information source to SIMS. Loom provides both a language and an environment for constructing intelligent applications. It combines features of both frame-based and semantic network languages, and provides some reasoning facilities.

The BNF syntax for the SIMS query language is shown in Figure 2.

```
<query> ::= (sims-retrieve <variable> | ({<variable>}+) <query-expr>)
<query-expr> ::= ({:and | :or} {<query-expr>}+)
<clause> ::= <concept-exp> | <relation-exp> | <assignment-exp> | <comparison-exp>
<concept-exp> ::= (<concept-name> <variable>)
<relation-exp> ::= (<relation-name> {<bound-variable>} {<term>})
<assignment-exp> ::= (:= <unbound-variable> {<arith-exp> | <set-exp>})
<set-exp> ::= ({<constant>}+)
<comparison-exp> ::= <member-comparison> | <arithmetic-comparison>
<member-comparison> ::= (member <bound-variable> <set-exp>))
<arithmetic-comparison> ::= (<comparison-op> {<arith-exp>} {<arith-exp>})
<arith-exp> ::= <number> | <bound-variable> | (<arith-op> <arith-exp> <arith-exp>)
<arith-op> ::= + | - | * | /
<comparison-op> ::= = | > | < | >= | <= | !=
<concept-name> ::= <symbol>
<relation-name> ::= <symbol>
<term> ::= <constant> | <variable>
<variable> ::= <bound-variable> | <unbound-variable>
<bound-variable> ::= ?<symbol>
<unbound-variable> ::= ?<symbol>
<constant> ::= <number> | <string>
```

Figure 2: BNF for the SIMS Query Language in LOOM Syntax

The following are the basic forms of a SIMS query:

```
(sims-retrieve ?v <query-expr>)
(sims-retrieve (?v1 ... ?vn) <query-expr>)
```

The variables listed after the **sims-retrieve** command, $?v$ and $?v_1 \ldots ?v_n$, are considered output variables. This means that the values of these variables are returned as the output of the query. All variables must be named with the prefix '?'. The query expression is composed of clauses and constructors. Clauses determine the values of the variables by binding the variables to specific types of values. In other words, clauses constrain the values of the variables. There are four types of clauses supported by the SIMS language which will be described in the next section. Clauses can be grouped by constructors into queries. Currently, the constructors provided are :and and :or.

A SIMS query returns as output a list of instantiations of the output variables which satisfy the bindings of the clauses in the query body. The following shows an example of output from a SIMS query.

```
(sims-retrieve (?name)  (:and (American-Large-Seaport ?seaport)
                              (port-name ?seaport ?name)))
```

```
==> (("Long Beach") ("New York") ("Norfolk") ...)
```

In this query the output variable ?name is bound to the values of the role port-name of American-Large-Seaport.

2.1.1 Clauses

Clauses are expressions that constrain the values which can be bound to a variable. A clause is satisfied when there exists values that satisfy the constraints on the variables in that clause. The following are the four types of clauses:

- Concept expressions:

```
(<concept-name> <variable>)
```

  where <concept-name> is the name of a concept, the variable is bound to an instance of the concept <concept-name>. An example of a concept expression is:

```
(Seaport ?seaport)
```

  This constrains the variable ?seaport to only the instances of the concept Seaport. Variables in concept expressions cannot be returned by the system.

- Relation expressions:

```
(<relation-name> <bound-variable> <term>)
```

  where <relation-name> is the name of a relation, <bound-variable> is a variable from a concept expression while <term> can be either a variable or a constant (a number or a string). The first clause states that there is a binary relation <relation-name> between <bound-variable> and <term>. The following are examples of this type of relation expression:

```
(port-name ?seaport ?name)
(seaport-country-code ?portCountryCode 'A123)
```

  The first expression is only satisfied if the value for ?name is the port-name of ?seaport. The second expression is only satisfied if 'A123 is the seaport-country-code of ?portCountryCode.

- Assignment expressions:

```
(:= <unbound-variable> <arith-expr>)
```

  This clause assigns to the unbound variable the computed result of <arith-expr>.

  For the following example, suppose we have a concept Seaport and its relations to its name (port-name) and to its number of cranes code (cranes). The following query will return a list of the names of a pair of seaports that have more than five cranes in total.

```
(sims-retrieve (?portname1 ?portname2)
               (:and (Seaport ?seaport1)
                     (Seaport ?seaport2)
                     (port-name ?seaport1 ?portname1)
                     (port-name ?seaport2 ?portname2)
                     (cranes ?seaport1 ?cranes1)
                     (cranes ?seaport2 ?cranes2)
                     (:= ?totalcranes (+ ?cranes1 ?cranes2))
                     (> ?totalcranes 5)))
==> (("Long Beach" "Norfolk")
     ("New York" "San Diego")
     .
     .
     .
     )
```

- Comparison expressions are used to express a constraint on variables. The following are forms of member comparisons:

64

```
(member <bound-variable> <set-exp>)
```

where a <set-exp> is defined as a set of constants. This clause is satisfied if the variable is bound to one of the constants (i.e., strings or numbers) in the <set-exp>.

The following are examples of member comparisons:
```
(member ?name ("Long Beach" "San Diego" "Newport Beach"))
```

This expression is only satisfied if the value for ?name matches one of the three strings in the set.

Another type of comparison expression uses the arithmetic comparison operators: =, >, <, >=, <=, !=.
```
(<comparison-op> <arith-expr> <arith-expr>)
```

The following are examples of the arithmetic comparison:
```
(> ?cr 5)
(= ?depth 120)
```

The first example checks that the the number of cranes (?cr) of a seaport is greater than five. The second example verifies the channel depth (?depth) of a seaport is equal to 120.

2.1.2 Query Expression Constructors

This section describes the two expression constructors supported by SIMS.

(:and $expr_1$ ...$expr_n$) — CONJUNCTION

> This returns the values for which each of the expressions $expr_j$ is satisfied.
> Example: (:and (Seaport ?x) (port-name ?x ?y))
> This expression is satisfied if ?x is a Seaport and ?y is the name of that seaport.

(:or $expr_1$ ...$expr_n$) — DISJUNCTION

> This returns the values for which at least one of the expressions $expr_j$ is satisfied.
> Example: (:or (Small-Seaport ?x) (American-Large-Seaport ?x))
> This expression is satisfied if ?x is either a Small-Seaport or an American-Large-Seaport.

2.2 SQL Syntax

SIMS also accommodates queries written in a subset of SQL syntax. A query in SQL syntax is first translated into the native Loom query language and then processed by SIMS internally. This SQL-syntax front end is different from a typical SQL query engine, such as a relational database, in two important ways.

- Syntax: SIMS' SQL front end accepts only a subset of standard SQL, a subset which easily corresponds to the internal Loom query language variant used in SIMS.

- Semantics: SIMS' SQL front end uses SQL to refer to SIMS domain concepts and relations, which are high level source-independent descriptions ("views") of the application domain. The terms do not necessarily refer to tables in any particular database.

The BNF syntax for the SIMS query language is shown in Figure 3.

In both SELECT lists and constraint conditions, attributes must always be specified using the fully qualified (Concept.attribute) syntax, even if only a single concept is referenced. This is because parsing of the SQL might take place in an environment where the schema of the underlying view might not be available, so there might be no context providing a way to assign attributes to concepts. The following is a correct example:

```
<query>   ::= SELECT <ret-param>{,<ret-param>}+
              FROM <concept-spec>{, <concept-spec>}+
              WHERE condition {, condition}+
<ret-param>   ::= <colname>  | <expr>
<colname>  ::= <CONCEPT-NAME>.<ATTRIBUTE-NAME>  | <ALIAS>.<ATTRIBUTE-NAME>
<concept-spec>  ::= <CONCEPT-NAME>  | <CONCEPT-NAME> <ALIAS>
<expr>  ::= (<expr> {,expr}+)  |
              <constant>  |
              -<expr>  | +<expr>  |
              <expr> <op> <expr>
<constant>  ::= <NUMBER>  | <STRING>  | NULL
<op>  ::= *  | +  | -  | /
<comparison ops>  ::= =  | !=  | <>  | >  | <  | >=  | =<
<condition>  ::= <expr> <comparison-ops> <expr>  |
                 <expr> IN <expr>  |
                 <expr> LIKE <expr>  |
                 <condition> AND  | OR <condition>  |
                 NOT <condition>  |
                 (<condition>)
```

Figure 3: BNF for the SIMS Query Language, SQL Syntax

```
SELECT ConceptZ.a
FROM ConceptZ
WHERE ConceptZ.b > 10
```

while the next two examples are incorrect because attributes are not specified with the fully qualified syntax.

```
SELECT a
FROM ConceptZ
```

```
SELECT ConceptW.b
FROM ConceptW
WHERE b like "%LARGE%"
```

As alluded to above, aliases can be used if desired:

```
SELECT R.a, R.b, S.b, S.c
FROM ConceptX R, ConceptY S
WHERE R.d = S.e
```

However, Concept.* (meaning all attributes) is not supported. In addition, SIMS does not currently distinguish between sets and bags of tuples. Practically, this means that in SELECT statements everything is distinct.

The following is an example of SIMS query in SQL format

```
SELECT Seaport.port-name
FROM Seaport
WHERE Seaport.cranes > 7
==> ("Long Beach" "Norfolk" ...)
```

This query asks for the names of large seaports. Equivalently, the following query returns the same information.

```
SELECT Large-Seaport.port-name
FROM Large-Seaport
```

There are a few exceptions in SIMS' SQL support. Constraints and expressions are currently only expressed in terms of simple arithmetic and boolean operators. In addition, SIMS' treatment of aggregate operations is currently very limited. Specifically, the following are not supported:

- Nested SELECTs.

66

- START WITH, GROUP BY, HAVING, CONNECT BY conditions.

- ORDER BY, FOR UPDATE.

- set operations UNION, UNION ALL, INTERSECT, MINUS.

- use of ROWNUM or other pseudocolumns.

Finally, as a (Lisp) syntactic convenience, the system is configured by default to intepret any occurrence of the underscore character (_) in concept names and attribute names to the dash (-) character. For example, the concept-name RUNWAY_LENGTH would be rendered as RUNWAY-LENGTH. This detail is unimportant except for users attempting to match up with a preexisting SIMS domain model using terms denoted with the (-) character.

3   The Domain Model

A domain model provides the general terminology for a particular application domain. This model is used to unify the various information sources that are available and provide the terminology for accessing those information sources. Throughout this manual we use a simple application domain that involves information about various types of seaports. The example is simple so that we can provide a complete, but short, description of the model. Figure 4 shows our example domain model.

3.1   The Model: Classes and Attributes

The domain model is described in the Loom language, which is a member of the KL-ONE family of KR systems. In Loom, objects in the world are grouped into "classes". Our example domain has several classes: Seaport, Large Seaport, Small Seaport, American Large Seaport, European Large Seaport and Country. The classes are indicated with circles in Figure 4. *Subclass* relationships are shown by dark solid arrows. For example, the class Large Seaport is a subclass of Seaport. This means that every instance of Large Seaport is also an instance of Seaport. A class can have any number of subclasses, but (currently) SIMS allows a class to have at most one superclass.

The figure also shows that Large Seaport and Small Seaport form a *covering*. This means that the class Seaport is the union of Large Seaport and Small Seaport. Thus, every seaport is either a large seaport or a small seaport.

Classes generally have *attributes* associated with them. For instance, the class Seaports has six attributes associated with it, a geographic code (geoloc-code), a port name (port-name), the number of cranes in the port (cranes), the channel depth (depth), the seaport's country code (seaport-country-code), and a country (country). This means that every seaport has a corresponding geographic code, port name, number of cranes, depth, and country code. Attributes are *inherited* down to subclasses. Thus, every large seaport will also have these six attributes, since Large Seaport is a subclass of Seaport. European large seaports have seven attributes, since they inherit the six attributes from Large Seaport, plus there is an additional attribute, the tariff code, associated with that class.

Classes can be defined as either primitive classes or they can be defined in terms of other classes. A primitive class has no explicit definition specifying the constraints that differentiate it from its superclass. For example, one might could create the class Large Seaport without specifying what constraints differentiate it from its superclass, Seaport. In terms of modeling a set of sources, this is useful in the case where you have two sources, where one is clearly a subclass of the other, but there is no simple way to characterize the specific subclass of information it contains.

Alternatively, it is possible to define the relationship between a subclass and superclass by explicitly describing the constraints on the subclass. For example, a large seaport might be defined as a seaport with more than seven cranes. This is what we have done in our example domain, as shown in Figure 4, where the class Large Seaport is defined as Seaport $\wedge$ (> cranes 7).

3.2   Specifying the Model: Class Definitions

Figure 5 shows the six class definitions that must be given to SIMS to specify the classes in our example domain. (See Figures 7 and 8 for a BNF description of the modeling language.) The class definition indicates whether or not the class is primitive. When a class is defined in terms of other classes, such as Large Seaport, the definition is specified using an "is" clause. The "is" and the "is-primitive" clauses are also used to indicate any attributes associated with the class.

Class definitions also have an "annotations" field. SIMS requires that every class has at least one defined *key*, which consists of one or more attributes that uniquely identify each instance in a class. Since there may be more than one way to uniquely identify an instance, a class can have multiple keys. For example, any seaport can be uniquely identified either using the geoloc-code or the port-name. Because more than one attribute may be necessary to uniquely identify an instance, a key can include multiple attributes. For instance, in another domain, it might be that street number, street name and city name are all necessary to uniquely identify a particular house.
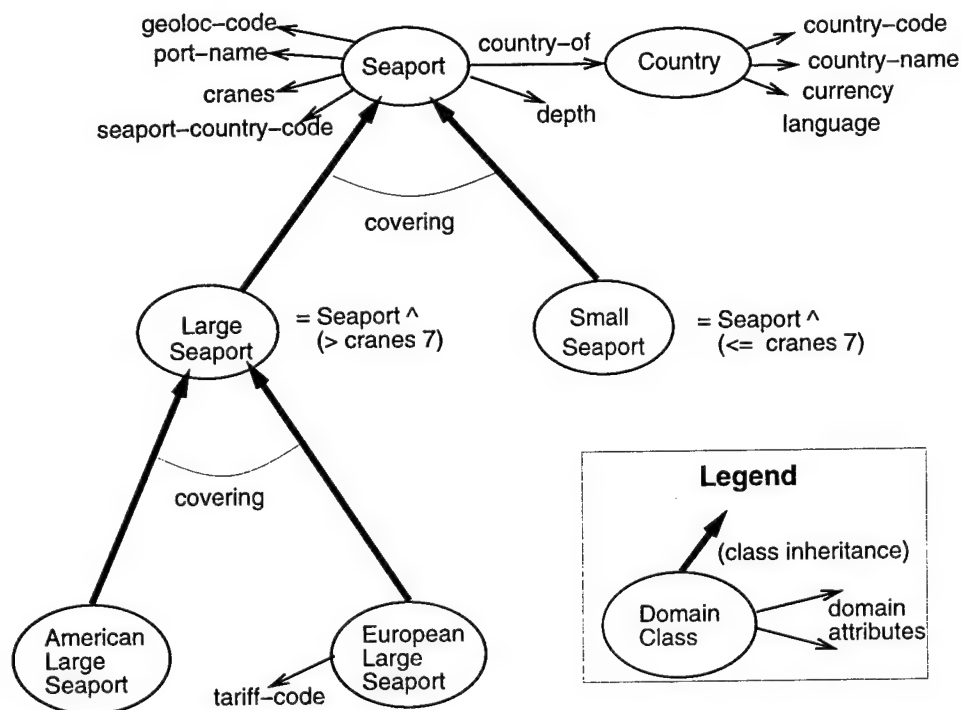
Figure 4: Domain Model

The annotations field of a class definition is also used to indicate that a class is a covering (i.e., the union) of some of its subclasses. For example, the class Seaport is the union of Large Seaport and Small Seaport.

3.3 Specifying the Model: Attribute Definitions

There are two types of attributes in SIMS. Most of the attributes in our example domain are *simple attributes*, in that they are basic classes: strings or numbers. But attributes can also represent relations between two defined classes. For instance, Seaport has an attribute called Country-of, so that every seaport is associated with a country. Thus, Country-of is a relation between Seaport and Country.

Figure 6 shows the relation definitions that define the attributes used in the domain model. Notice that each attribute has a domain and range. Defined relations (attributes that relate two classes) have a definition. For instance the Country-of relation has a definition which specifies that the relation holds between a seaport and a country if the seaport's seaport-country-code matches the country's country-code.

There cannot be two different attributes with the same name. In our example, Seaport has the attribute seaport-country-code, and Country has an attribute country-code. Even though these are both 'country codes', the names of the attributes must be different.[1]

The domain model is used as the basis for the SIMS query language that enables the user to construct queries. The classes included in the domain model are not necessarily meant to correspond directly to objects described in any particular information source. The domain model is intended to be a description of the application domain from the point of view of someone who needs to perform real-world tasks in that domain

---

[1]Had we wanted a Seaport to have an attribute called country-code, we would have done so only by making the model more complex. For instance, we could have created a class Geographic entity with an attribute country-code, which could have been a superclass of both Country and Seaport, in which case the attribute country-code would have been inherited down to both of these classes.

```
(def-sims-concept seaport
    :is-primitive (:and sims-domain-concept
                        (:the country-of country)
                        (:the geoloc-code string)
                        (:the seaport-country-code string)
                        (:the port-name string)
                        (:the cranes number)
                        (:the depth number))
    :annotations ((key (geoloc-code))
                  (key (port-name))
                  (covering (large-seaport small-seaport))))

(def-sims-concept large-seaport
    :is (:and seaport
              (> cranes 7))
    :annotations ((key (geoloc-code))
                  (key (port-name))
                  (covering (american-large-seaport
                             european-large-seaport))))

(def-sims-concept small-seaport
    :is (:and seaport
              (<= cranes 7))
    :annotations ((key (geoloc-code))
                  (key (port-name))))

(def-sims-concept american-large-seaport
    :is-primitive large-seaport
    :annotations ((key (geoloc-code))
                  (key (port-name))))

(def-sims-concept european-large-seaport
    :is-primitive (:and large-seaport
                        (:the tariff-code string))
    :annotations ((key (geoloc-code))
                  (key (port-name))))

(def-sims-concept country
  :is-primitive (:and sims-domain-concept
                      (:the country-code string)
                      (:the country-name string)
                      (:the currency string)
                      (:the language string))
  :annotations ((key (country-code))))
```

Figure 5: Class Definitions for our Example Domain

and/or to obtain information about it. SIMS is designed to allow users to query the domain model without specific knowledge of the way the actual information sources relate to the domain model. The next section describes how application developers describe the actual information sources and their relationship to the

domain model.

```
;;; Seaport attributes

(def-sims-relation geoloc-code
    :domain seaport
    :range string)

(def-sims-relation port-name
    :domain seaport
    :range string)

(def-sims-relation cranes
    :domain seaport
    :range number)

(def-sims-relation depth
    :domain seaport
    :range number)

(def-sims-relation seaport-country-code
    :domain seaport
    :range string)

(def-sims-relation country-of
    :domain seaport
    :range country
    :is (:satisfies (?s ?c)
          (:and (seaport ?s)
                (country ?c)
                (seaport-country-code ?s ?cc)
                (country-code ?c ?cc))))


;;; European Large Seaport attributes

(def-sims-relation tariff-code
    :domain european-large-seaport
    :range string)


;;; Country attributes

(def-sims-relation country-code
    :domain country
    :range string)

(def-sims-relation country-name
    :domain country
    :range string)

(def-sims-relation currency
    :domain country
    :range number)

(def-sims-relation lang
    :domain country
    :range number)
```

72

Figure 6: Attribute Definitions for our Example Domain

```
class-definition ::=
   (DEF-SIMS-CONCEPT ClassName
         is-clause
         annotations-clause)

is-clause ::=
   :IS-PRIMITIVE (:AND SuperClassName attr-clause*)  |
   :IS (:AND SuperClassName constraint-expr*)

annotations-clause ::=
   :ANNOTATIONS (annotation+)

annotation ::=
   (KEY (AttributeName+)) |
   (COVERING (SubClassName SubClassName+))

attr-clause ::=
   (:THE AttributeName ClassName) |
   (:THE AttributeName STRING) |
   (:THE AttributeName NUMBER)

constraint-expr ::=
   (test Term Term) |
   (:FILLED-BY AttributeName Term) |
   (:NOT-FILLED-BY AttributeName Term)

test ::=
   >  |  <  |  >=  |  <=  |  !=  |
```

Figure 7: BNF for Class Definitions

```
simple-relation ::=
  (DEF-SIMS-RELATION RelationName
     :DOMAIN ClassName
     :RANGE [NUMBER | STRING])

defined-relation ::=
  (DEFRELATION RelationName
     :DOMAIN ClassName
     :RANGE ClassName
     :IS (:SATISFIES (VariableName VariableName) constraint-expr))

constraint-expr ::=
   (:FOR-SOME (VariableName) constraint-expr)
   (:AND constraint-expr+) |
   (AttributeName Term Term) |
   (ConceptName Term) |
   (test Term Term)

test ::=
   >  |  <  |  >=  |  <=  |  !=  |
```

Figure 8: BNF for Attribute Definitions

4 Defining Information Sources

In order to extract and integrate data from an information source, a person building an application must describe the contents of the source using terms from the domain model and define the details of how the source is accessed. Each of these issues is addressed in turn.

4.1 Describing the Contents of an Information Source

Each information source is incorporated into SIMS by describing the data provided by that source in terms of the domain model presented in the previous section. This description provides the following information:

- The precise class of instances provided by a source.

- The set of attributes that are available from the source.

- The name of the source that provides the data (the next section will define additional information about accessing each source)

- The mapping from the table/class name of the source and the name used in the domain model.

- The mapping from the attribute names used in the source and those used in the domain model.

To illustrate the principles involved in representing an information source within SIMS, consider how a set of sources would be represented using the domain model described in the previous section. Figure 9 shows the example domain model linked to a set of seven separate sources.
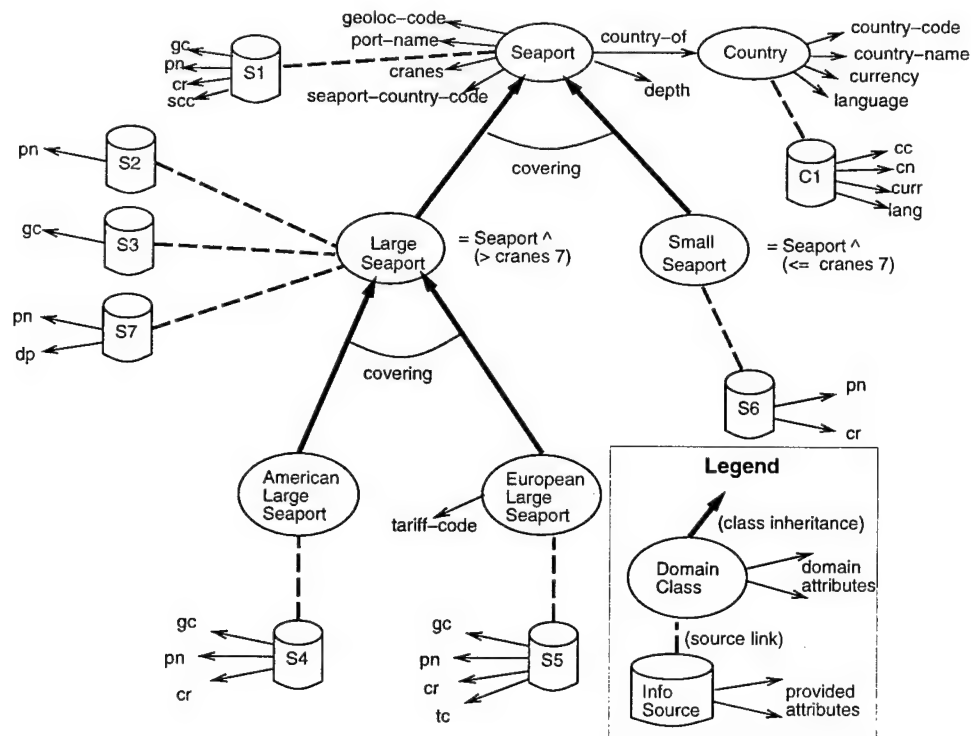
Figure 9: A Set of Sources Described by a Domain Model

In the figure, each source is linked to a class with a dashed line. The meaning of such a link is that the source provides exactly the set of instances described by the class of the domain model. Thus, the figure shows that S2, S3, and S7 all provide *exactly* the same set of large seaports. If there is another source that provides only a subset of a class of instances, then a new subclass in the domain model would be created

74

and the source would be linked to that class. Sources S4 and S5 are both examples of sources that provide subclasses of large seaports and thus are linked to the appropriate subclasses in the domain model.

Since different sources often provide different attributes for the same class, we do not require that all sources provide all attributes of a class. In the figure, the attributes provided by each source are shown next to the individual sources.

The general form of a source description statement is shown in Figure 10. There will generally be one of these statements for each table or relation in a source. However, in some cases, sources can be more naturally modeled by mapping a single relation in the source into more than one domain class. As shown in the figure, a domain class is used to describe a source table and DB and the domain attributes are linked to the corresponding attributes of the source.

```
(source-description <domain-class> <source-table> <source-db>
    (<domain-attribute-1> <source-attribute-1>)
    (<domain-attribute-2> <source-attribute-2>)
    ...
    (<domain-attribute-n> <source-attribute-n>))
```

Figure 10: General Form of a Source Description

Consider how a specific source in Figure 9 would be described. For source S7, which provides the port name and depth of Large-Seaport. The description of this source is shown in Figure 11.

```
(source-description Large-Seaport S7 EXKB7
    (port-name pn)
    (depth dp))
```

Figure 11: Source Description for Large-Seaports table of S7 Database

4.2 Accessing an Information Source

In addition to specifying the contents of an information source, the system also needs to know what information sources are currently available and how to access them. This section describes the basic commands for declaring information sources.

To make an information source available to the system, the name, and host must be declared in advance. This provides the information required for accessing an information source. The template for declaring an information source is shown in Figure 12. The <sims name> provides a term for referring to a specific information source with SIMS. The <source type> defines the specific type of source. The set of possible types are listed below. The :host is the name of the machine where the information source is running. The :agent-name of a source is the unique name that KQML uses to identify the particular source. Each SIMS agent also has a unique name and can serve as a source to other agents. The :db-name is the internal database name, which might not be unique since you may have multiple instances of the same information source running on different hosts. The :userid is the internal userid for a database.

```
(define-source <sims name> <source type>
      :host <information source host>
      :agent-name <name used by KQML>
      :db-name <name used within a DB>
      :userid <user id for a DB>)
```

Figure 12: Template for Defining an Information Source

There are currently four predefined source types and they are explained in the subsections below.

4.2.1 loom-kb-source

For small sources, such as translation tables or source tables of data, it is very simple to create a local knowledge base. In Section 9, we provide an example that uses a number of KBs. This type of source is referred to as a `loom-kb-source`. For example, to define the source called `mapping-table` as a knowledge base, it would be declared as follows:

```
(define-source mapping-table loom-kb-source)
```

4.2.2 kqml-odbc-sql-source

The most frequently used type of source will probably be a `kqml-odbc-sql-source`. This is a source that supports the full SQL query language and supports ODBC communication. This can be used to communicate with any of the commericially available relational database systems, although it may require purchasing a separate ODBC package from either the database vendor or a third-party vendor. An example of how such a source would be declared is shown below:

```
(define-source giti8i kqml-odbc-sql-source
               :host "isd54.isi.edu"
               :agent-name "sql_server"
               :db-name "assets"
               :userid "abc")
```

4.2.3 kqml-sp-sql-source

In some cases a user may build their own source that supports a restricted set of SQL. For these purposes, we have defined a `kqml-sp-sql-source`, which is a source that communicates using KQML, but only supports selections and projections (SP) within the SQL language. Such a source would be declared as follows:

```
(define-source URL_SRC kqml-sp-sql-source
               :host "isd54.isi.edu"
               :agent-name "URL_SRC")
```

4.2.4 kqml-sims-source

Each instantiation of SIMS can be thought of as an information agent that can in turn provide data to other SIMS agents. To do this, you would declare the other SIMS agents as `kqml-sims-source`. These source use the SIMS query language and send information using KQML. Such a source would be declared as follows:

```
(define-source other-sims kqml-sims-source
               :agent-name "LOGISTICS")
```

4.3 Modeling Hints

A single domain class can be used to describe multiple sources. For instance, the domain class "Large Seaport" in the "Seaports domain" is linked to the sources S2, S3, and S7.

Note that if two (or more) sources (e.g., S2 and S7) are linked to the same domain concept (e.g., `Large-Seaport`) and those sources have a common attribute (e.g., `port-name`) then both sources need to have the same values of that attribute (i.e., every port name, pn, present in S2 also has to be present in S7, and vice versa). A source concept may contain attributes that are not linked to any domain relation.

Consider the example shown below. In this example there is only one source (S1) linked to the concept "Seaport".

76

geoloc-code
port-name
cranes
Seaport
gc
pn
cr
scc
S1
seaport-country-code

Large
Seaport
= Seaport ∧
(> cranes 7)

covering

American
Large
Seaport

European
Large
Seaport
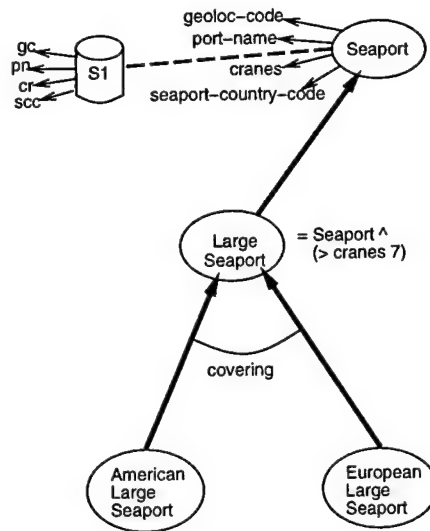
- The query "return all port names of Large Seaports" is "answerable" because "Large Seaport" is defined as the subset of "Seaport" that has at least 7 cranes. As the source S1 provides all seaport names, we can select (from S1) only those port names that have at least seven cranes.

- The query "return all port names of American Large Seaports" is "not answerable" because there is no way of specifying which Large Seaports are American.

For another example, consider the model below:

geoloc-code
port-name
cranes
Seaport
seaport-country-code

Large
Seaport
= Seaport ∧
(> cranes 7)

covering

American
Large
Seaport

European
Large
Seaport

pn
S4
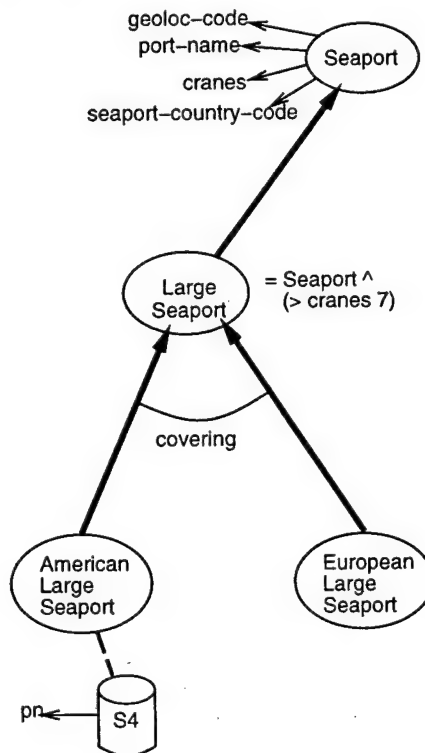
Here there is only one source (S4), linked to the concept "American Large Seaport". The query "return all port names of Large Seaports" is "not answerable" because we have information only about "American Large Seaports", not ALL Large Seaports.

Finally, consider a third example:

77
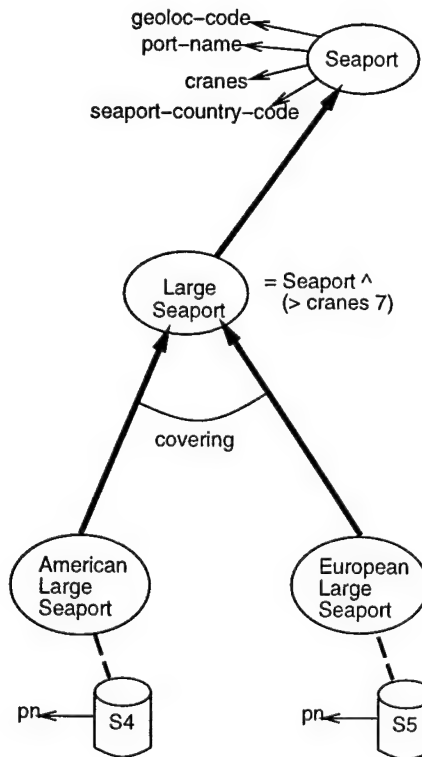
Here there are two sources S4 and S5. The query "return all port names of Large Seaports" is "answerable" because "American Large Seaport" and "European Large Seaport" represent a covering of "Large Seaport". Consequently, the union of port names from S4 and S5 represents all "port names of Large Seaports".

5 Information-Source Wrappers and Communication

Once the SIMS planner has selected the desired sources for a user's query and devised a plan for obtaining the required information, it must communicate with the individual information sources. Sometimes the information source may be complex and difficult to communicate with and additional data processing or functionality may be required. In order to modularize this process and cleanly separate query planning from communication issues, SIMS requires that for each type of information source there exist a wrapper with which it will communicate. The purpose of the wrapper is to mediate between SIMS and the information source. The wrapper must be capable of translating between the query language obtained from SIMS and the information source's query language if necessary, as well as translating between the data output format of the information source and a format appropriate for SIMS.

This section explains how wrappers are used by SIMS. The first subsection describes the data that is communicated. The subsequent subsections describe the protocols by which the data is passed; KQML and CORBA.

5.1 Information Source Wrappers

An information source's wrapper will receive a query from SIMS as input. The syntax of this query language can be varied, so long as the wrapper and SIMS have agreed upon it. Predefined examples include the SIMS query language or SQL. See section ?? for more on information sources. One restriction is that all concepts and roles used in the query will be drawn *only* from that information source. Note that at the time when such communication takes place SIMS has already determined that the query being sent to the information source can be processed in its entirety by that source alone.

The information source's wrapper performs any necessary mediation between SIMS and the information source. This may involve translating the query into the information source's particular query language, providing additional information to the information source or any other necessary reconciliation between SIMS and the information source. It then submits a query to the source and retrieves the data. Next, it packages this data into a list of tuples corresponding to the variable parameters used in the submitted query. This tuple is then returned to SIMS. See Figure 13.
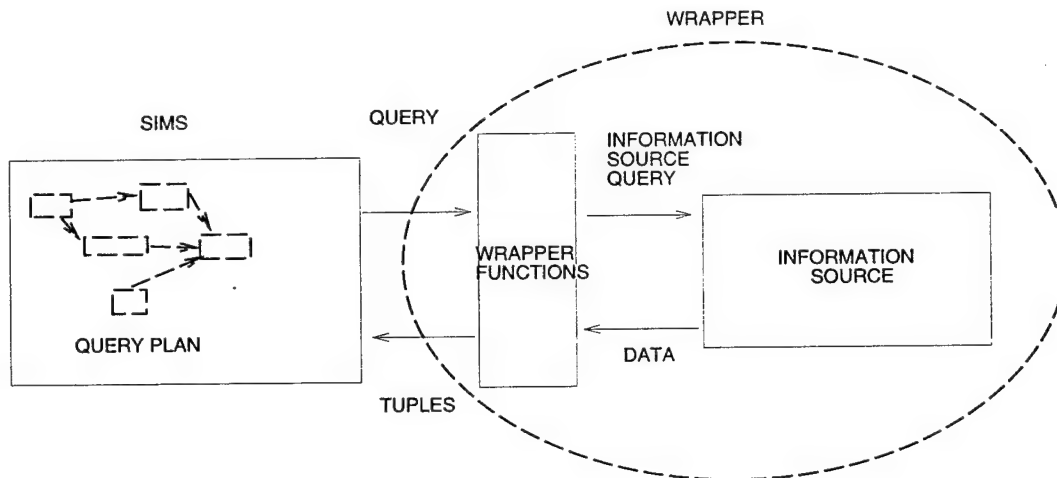

Figure 13: Data Flow between SIMS and Wrappers

In this way, SIMS is insulated from the particulars of each information source. All the complexity of an information source is hidden from SIMS via the wrapper module.

5.2 Remote Communication Using KQML

If an information source server is loaded into the running SIMS environment, a straight function call to the appropriate information source wrapper function is sufficient to process a query. For communication with servers running on remote hosts, SIMS uses the Knowledge Query and Manipulation Language (KQML) protocol, which is a language for communication and knowledge sharing between autonomous programs. A simplified view of KQML-based communication is presented in Figure 14.
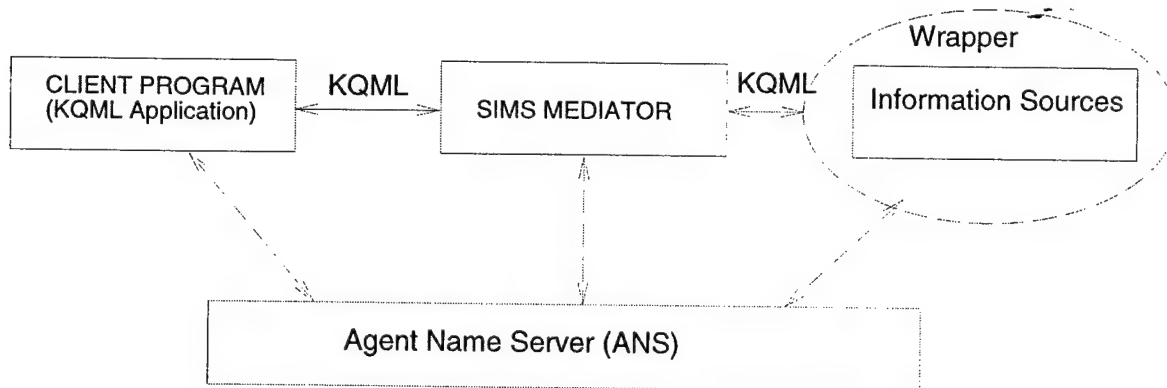
Figure 14: Communication via KQML

For our purpose, KQML provides two main types of functionality that ease the communication between clients and servers (KQML refers to both clients and servers as *agents*). KQML provides a flexible standard language for client-server communication that is available for many platforms, as well as implementation in different languages. It also provides a registry of all clients and servers, so that a client only need to refer to the name registered on the registry by the server (which is usually the name of the service provided and hence more meaningful than just a host address) to communicate with the servers.

The central registry of services in KQML is called the *agent name server (ANS)*, and it records all KQML agents and their addresses. We are mostly interested in the ANS for providing the addresses of information sources SIMS needs to communicate with (this address resolution process happens transparently and does not require user intervention). The client and server must both be registered with the ANS. The global variable kqml::*kqml-ans-host* specifies where the ANS is located, and both the client and server should agree on an ANS accessible to both. A server registers itself by executing the command:

```
(KQML:START-KQML <server name>)
```

where <server name> must be unique. Clients are started by invoking (start-kqml-client), and this function call will register the client using a unique name of the following form, <user>@<host>-<timestamp>, where timestamp is obtained from (get-universal-time).

In order to check what is available on a ANS, or to verify that a service that was registered is up, one can issue the following command in a UNIX shell:

```
($KQML_HOME)/bin/agentls
```

Note that the KQML clients/servers only contact the ANS once to verify the existence of a server and to get its address. The user need not know where a particular server is located but only its name (e.g., SQL-QUERY-SERVER). KQML transparently resolves the location through the ANS and caches each resolved location. The communication protocol used by KQML is TCP/IP. KQML creates a process that listens on a remote TCP/IP stream in order to detect messages from remote hosts.

The ANS used by KQML must be accessible to both SIMS and to any users of the SIMS system, but need not be run on those systems itself. To run a ANS at a site, execute the following command in a Unix shell:

80

```
($KQML_HOME)/bin/startans <hostname> &
```

The client must know the messages supported by the server program because only those can be processed. In KQML terms, SIMS acts as a mediator between the client and the information sources. A KQML mediator receives a request and either delegates it to one or more other servers, or processes it internally/locally (e.g., in a local database). Hence the information source server needs to define a handler for the messages it will support and the client needs to know these messages together with their form.

SIMS currently uses only the `:ASK-ALL` KQML performative to communicate with remote information source servers. The KQML message sent by SIMS to the information source servers are of the form:

```
(:ASK-ALL :SENDER <SIMS server> :RECEIVER <info-source server>
          :REPLY-WITH T :CONTENT (<SQL query><hostname:dbname><username>))
```

while the KQML message sent by a client program must be of the form:

```
(:ASK-ALL :RECEIVER <SIMS server>
          :REPLY-WITH T :CONTENT (SIMS-RETRIEVE <output args><query body>))
```

5.3 Remote Communication Using CORBA

As CORBA [3] is supported by virtually all the industry leaders, making SIMS a CORBA-compliant application broadens the area of potential SIMS application. For instance, any CORBA-compliant application is able to act like a SIMS client (i.e., to send queries to SIMS and to use the returned answers). Furthermore, a CORBA-compliant version of SIMS is able to access information sources that use CORBA-based software wrappers. A simplified view of CORBA-based communication is presented in Figure 15.

CORBA defines distributed services for inter-process and inter-platform messaging, and it provides interoperability between applications written in different programming languages, running on different machines in heterogeneous, distributed environments. CORBA is an interoperability standard that has several implementations (e.g., Orbix, ILU, VisiBroker), and we currently support the Orbix 2.2 [7] implementation of CORBA.

A CORBA-compliant interface is provided to let SIMS act both as a CORBA client (e.g., to access information sources via CORBA wrappers) and a CORBA server (i.e., to receive queries from CORBA-compliant applications).

Based on our CORBA-to-KQML adapter, any number of CORBA-compliant applications can use SIMS as a query-answering system. SIMS is unable to discern between the queries sent by CORBA-based and KQML-based clients, and, consequently, the communication will be done in a uniform, transparent manner. SIMS can also access information sources that use CORBA wrappers: each type of information source will offer a distinct, specific interface, and we will need a specially-designed KQML-to-CORBA adapter for each type of information sources. However, instead of being compelled to modify the SIMS code for each new type of information sources, we will be able to implement these adapters as stand-alone applications, and SIMS will be unable to tell a CORBA-based source from a KQML-based source.

5.3.1 The CORBA-to-KQML adapter

The idea behind the CORBA-to-KQML adapter is to create a CORBA server that provides services equivalent to a minimal subset of the API provided by KQML. The CORBA-to-KQML adapter has a dual nature: it acts like a CORBA server for the CORBA client, and once it receives a query, it turns it into a KQML-based query and sends it to SIMS (i.e., the adapter acts like a KQML-based client of SIMS). One important advantage of the CORBA-to-KQML adapter is that developers of the CORBA clients do not have to know
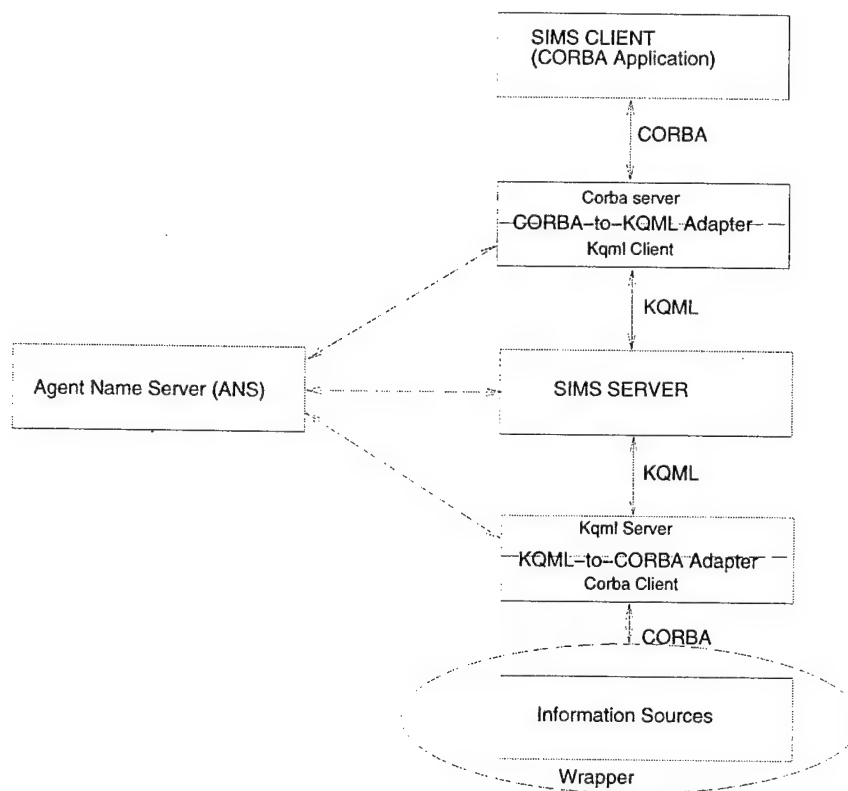
Figure 15: Communication via CORBA

anything about KQML. From their point of view, they are accessing a CORBA server with an easy-to-use interface [6].

Figure 16 shows a simple example of a CORBA client that reads a query specified as a command line argument and sends it to the SIMS server named `SimsAgent`. In order to communicate with the SIMS server, the client sends the query to the CORBA-to-KQML adapter named `CORBA2KQML` which is located on the machine `vigor.isi.edu`.

5.3.2 The KQML-to-CORBA adapter

As information sources can be accessed in different ways (e.g., different information sources may provide different IDL interfaces, a single information source might offer several IDL interfaces, etc.), it is not possible to design a unique KQML-to-CORBA adapter that can be used for any type of CORBA-based wrappers.

In order to avoid any changes in the SIMS code, we designed the adapter so that SIMS sends it the message as if it were addressed to an information-source wrapper. The message is received by the adapter, which translates it to a sequence of CORBA requests and asks the corresponding CORBA server to execute the resulting sequence of services. For details see [6].

82

```
main(int argc, char **argv)
{
  CORBA2KQML* anObj = NULL;
  char *query;

  if (argc>=2)  {
    query=argv[1];
  } else {
    printf("Usage: client querystring");
    exit(1);
  }

  TRY{
      anObj = CORBA2KQML::_bind(":CORBA2KQML","vigor.isi.edu",IT_X);
  }
  CATCHANY{
      cout << "error" << IT_X << endl;
  }
  ENDTRY

  TRY{
      c_kqml_message_var answer;
      char* ptr;
      char p[100];
      int i, n;

      fprintf( stderr, "ret_send_msg = %ld",
                  anObj->SendQuery( 1200, query, "SimsAgent", answer) );

  }
  CATCH(CORBA::SystemException &se){
      cout << "@@@@ #### CORBA_2_KQML exception raised!!!" << endl;
      cout << endl << &se <<endl;
  }

}
```

Figure 16: CORBA client

83

6 Running SIMS

This section describes the commands that can be used to interact with SIMS. The commands can be issued in two ways: 1) Through a command-line interface via the Lisp listener or 2) using the graphical interface.

6.1 Top-Level Commands

The top level commands of SIMS can be classified in these groups: query commands, query set management commands, information source management commands, and tracing commands. They are described in the following sections.

6.1.1 Query Commands

(sims-retrieve <parameter-list> <query-exp>)

This is the command to execute a query. A complete description of the query syntax is provided in Section 2. This command will cause data to be retrieved and displayed.

(run-query <num>)

This command has the same effect as (sims-retrieve). However, rather than receiving a literal query as an argument, it causes the pre-stored query denoted by <num> to be executed. (see query set management subsection).

6.1.2 Query Set Management Commands

Sometimes it is convenient to have frequently used queries stored in the system. A query set can be predefined by setting the global variable *queries* to the list of queries. Often, this query set is predefined in a file which can be loaded at run-time. This query set can also be used from the graphical interface described in the next section.

(list-queries) Provides a list of the numbers of predefined queries.

(load-comment <num>) Retrieves the comment for query <num>.

(load-query <num>) Retrieves query <num> and displays it in normal form.

(plan-query <num>) Generates the plan for performing query <num>), but does not execute it.

(display-plan <plan>) Displays the plan returned by plan-query. (E.g., (display-plan (plan-query 1)) will display the plan for query 1.

(run-query <num>) Executes query <num>.

The syntax for the query set is:
```
(setq *queries* '(
  (<num> <comment> <query>)
  (<num> <comment> <query>)
  .
  .
  .
  .))
```
Here is an example of a query set:

```
(setq *queries*
  '(;; Large Seaport queries
```

```
(1 "List the geoloc-codes and number of cranes for all large seaports"
    (sims-retrieve (?gc ?cr)
      (:and (large-seaport ?ls)
            (gc ?ls ?gc)
            (cr ?ls ?cr))))

(2 "What is the currency of France"
   (sims-retrieve (?currency)
      (:and (country ?c)
            (cn ?c "FRANCE")
            (currency ?c ?currency))))
))
```

6.1.3   Information Source Management Commands

The commands for manipulating the information sources are:

(list-sources) Lists all of the declared information sources.

(available-sources) Lists all of the currently available information sources that the system can access.

(initialize-source <unique-id>) Initializes the given information source. In effect, this command tells SIMS that the source is ready to provide information and can be used in planning steps.

(initialize-all-sources) Initializes all defined information sources. This can also be referred to as `(init-all-is)`.

(close-source <unique-id>) Closes the given information source. The source is marked unavailable and no longer used in planning.

(close-all-sources) Closes all of the defined information sources.

6.1.4   Tracing Commands

In order to facilitate debugging and show the behavior of the system in a greater detail, the following commands instruct SIMS to print additional information about its processing.

(sims-trace-on) Turns on tracing. SIMS prints additional information on the query planning and execution, such as plan steps, partial reformulations, information sources accessed, intermediate results, etc.

(sims-trace-off) Turns off tracing.

(sims-trap-on) SIMS traps all errors (returning nil at the end of execution if the errors prevented the successful execution).

(sims-trap-off) When an error occurs in the processing, SIMS allows the original error handler to interrupt the execution. This command is useful when debugging an application.

6.2   Graphical User Interface

This section describes how to interact with SIMS through its graphical user interface.

The graphical interface to SIMS is invoked via two function calls, `(sims-java-server)` and `(java-interface)`, in that order. Below is a sample interaction:

```
SIMS(19): (sims-java-server)
#<MULTIPROCESSING:PROCESS Java Server @ #x1a203a2>
SIMS(20):
Java Listener started on port 6001 12:19:09 12/5/1997

SIMS(20): (java-interface)
;;; Spawning java interface...
SIMS(21):
;Accepting Java connection from [128.09.208.55]/6001
SIMS(21):
```
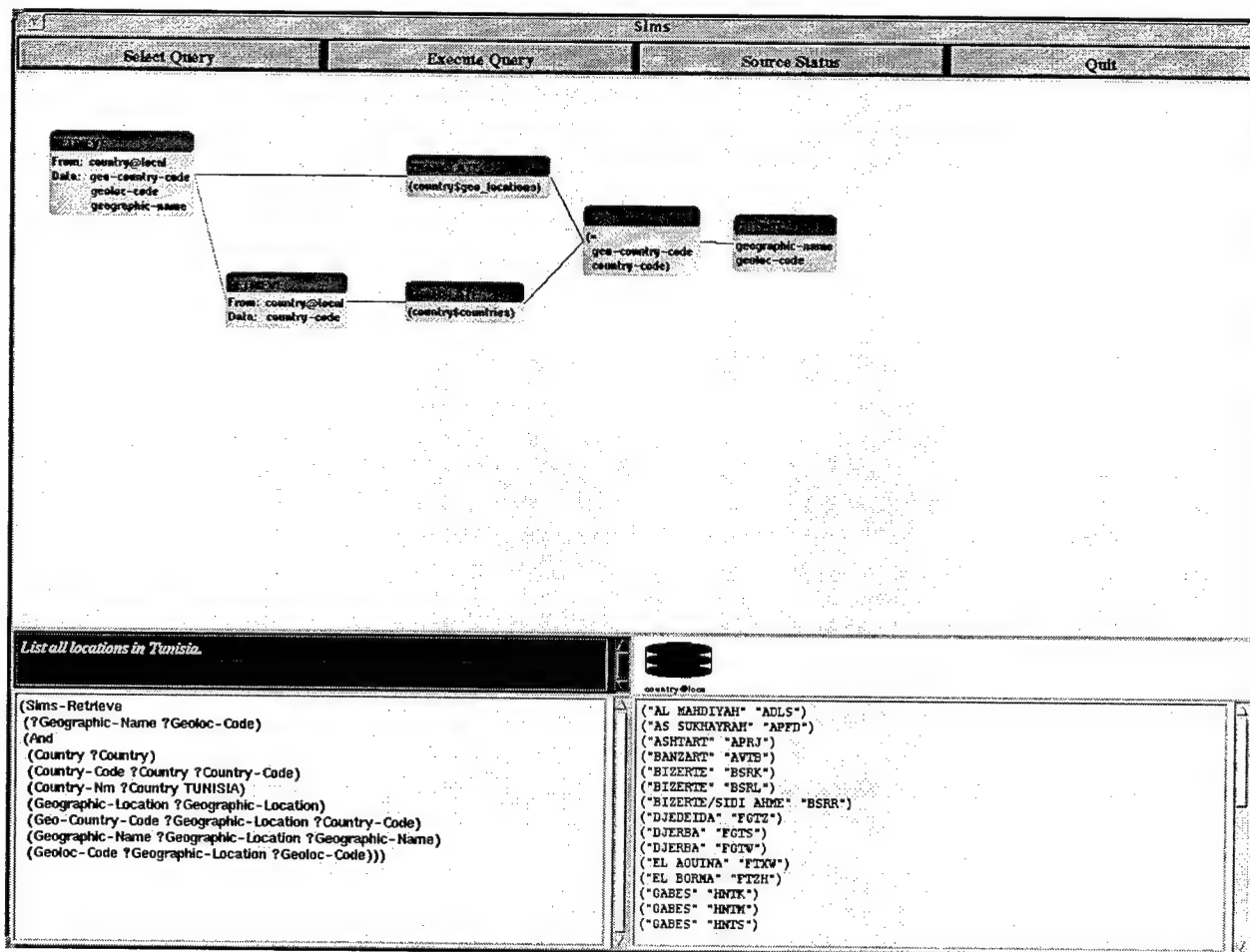

Figure 17: SIMS graphical interface

The SIMS interface (see Figure 17) is divided into four main panes: the Command Panel (top), the Interaction/Trace pane (lower right quadrant), the Query pane (lower left quadrant), and the Graph pane (upper half).

The user issues commands by selecting a command from the Command Panel. Typically, an interaction sequence will proceed as follows:

1. Click Select Query and choose a query to solve. The chosen query will be displayed in the Query pane.

2. To perform the actual retrieval, once a query has been chosen, click `Execute Query`. The appropriate plan graph will be shown in the Graph pane and the state of the execution is indicated by coloring the nodes in the graph. White denotes an unexecuted plan action, green denotes a currently executing plan action, grey denotes a completed plan action, and pink denotes a failed plan action. The final answer will be displayed in the Interaction/Trace pane.

6.2.1 Graphical Interface Commands

Select Query Brings up a menu of the set of queries that were loaded into the system on startup (in the variable `*queries*`). Each query is a checkbox that can be selected. Only one query can be selected at a time. The selected query will be displayed in the Query pane. This is the query that will be used by `Execute Query`.

Execute Query Executes the current query. The graph plan currently being executed will be shown in the graph panel. The final result will be displayed in the Interaction/Trace pane.

Source Status Brings up the list of existing information sources. Each source has a status associated with it, which the user may change manually. AVAILABLE denotes a source that is available to answer queries, UNAVAILABLE denotes a source that is unavailable to answer queries and therefore cannot be incorporated into a plan, FAULTY denotes a source that will cause a run-time fault. The FAULTY status can be chosen to force a source to fail and simulate a replanning situation.

Quit Exits the SIMS interface.

6.3 Plan Cost Evaluation

The SIMS architecture allows the user to change the policy of the generation of query access plans to account for different cost models. The function `set-evaluation-function` establishes the function that will guide this generation.

Currently, SIMS provides two functions. The first one, `ucpop::evaluate-plan-cost`, generates plans with the minimum number of steps. The second one, `ucpop::evaluate-plan-cost-by-size`, produces query plans in which the size of intermediate data transmitted from the information sources and processed in local joins is minimized. It uses a series of traditional database techniques to estimate the size of the queries. It considers both the expected number of tuples that a query will produce and the projection attributes. In order to calculate this estimate, it uses some statistics computed from the current contents of the information sources, such as, number of instances of a concept, number of distinct values (present in the source) of an attribute, and maximum and minimum values for numeric attributes.

Generally, `ucpop::evaluate-plan-cost-by-size` both improves the efficiency of the planning process (2 to 5-fold speed-up) and the quality of the generated plans. For complex queries this should be the function of choice. For simple queries the performance of both functions is similar.

In summary,

- to use `ucpop::evaluate-plan-cost` (the default), evaluate:

  > `(set-evaluation-function #'ucpop::evaluate-plan-cost)`

- to use `ucpop::evaluate-plan-cost-by-size`, evaluate:

  > `(set-evaluation-function #'ucpop::evaluate-plan-cost-by-size)`

7 Trouble Shooting

What do you do once you have built the wrappers for your information sources, defined the domain model and the information sources, and submitted the first query to SIMS only to find that it does not work? We recommend that you first test your system incrementally from the bottom up by testing the wrappers to the information sources, then testing the source-level queries, and finally testing your domain-level queries. This section describes each of these in turn.

7.1 Testing the Information-Source Wrappers

Before invoking SIMS, individual wrappers for all of the information sources that are to be used should be thoroughly tested. Each wrapper should accept a source-level query as input and return a set of tuples that represent the answer to that query. To test the individual wrappers, invoke the function (info-source-retrieve <source-object> <query>) for each wrapper. An example of *info-source-retrieve* invocation is the following:

```
(INFO-SOURCE-RETRIEVE 'EXKB5@LOCAL
          '(SIMS-RETRIEVE (?PN)
                  (:AND (EUROPEAN-LARGE-SEAPORT ?EUROPEAN-LARGE-SEAPORT)
                        (CR ?EUROPEAN-LARGE-SEAPORT ?CR)
                        (PN ?EUROPEAN-LARGE-SEAPORT ?PN)
                        (>= ?CR 15))))
```

where EXKB5@LOCAL is the *name* of the information source, and the second argument is a source level query. The result of this invocation is the following:

```
UCPOP Stats: Initial terms = 7 ;   Goals = 4 ;  Success (3 steps)
     Created 33 plans, but explored only 37
     CPU time:    0.0500 sec
     Branching factor:  1.000
     Working Unifies: 62
     Bindings Added: 20


(("Bristol") ("Rotterdam"))
```

If this wrapper does not return the expected data, one must determine the cause of the problem and fix it before continuing to the next step. Any error detected at this point implies that there is a problem with the wrapper, not with SIMS.

7.2 Testing the Source-Level Queries

Once all of the wrappers are working correctly, it is time to begin testing the source-level queries in SIMS. The first thing to test are exactly the same source-level queries that were used to test the individual wrappers. An example of testing a source level query is the following:

```
(SIMS-RETRIEVE (?PN)
     (:AND (EUROPEAN-LARGE-SEAPORT ?EUROPEAN-LARGE-SEAPORT)
           (CR ?EUROPEAN-LARGE-SEAPORT ?CR)
           (PN ?EUROPEAN-LARGE-SEAPORT ?PN) (>= ?CR 15)))

UCPOP Stats: Initial terms = 7 ;   Goals = 4 ;  Success (3 steps)
     Created 33 plans, but explored only 37
     CPU time:    0.0500 sec
```

Error
└── Simple Error
    └── Sims Error
        ├── Database Error   (DBNAME, HOSTNAME, USERID)
        │   ├── Unknown Password Error (PASSWORD–FILE)
        │   └── Oracle Error (RAWSTRING, PREFIX, ERRNO, MESSAGE)
        ├── Syntax Error
        │   ├── SQL Syntax Error (SQL)
        │   ├── Hostname Syntax Error (HOSTNAME)
        │   └── SIMS Syntax Error
        │   ├── Unrecognized Request Error (REQUEST)
        │   └── SIMS Query Syntax Error
        ├── Network Error
        │   └── Unknown Host Error (HOSTNAME)
        ├── Communications Error
        │   └── KQML Error (ANS–HOST, AGENT–NAME)
        ├── System Error
        │   └── Memory Allocation Error
        └── Driver Error
            ├── Unobtainable Error
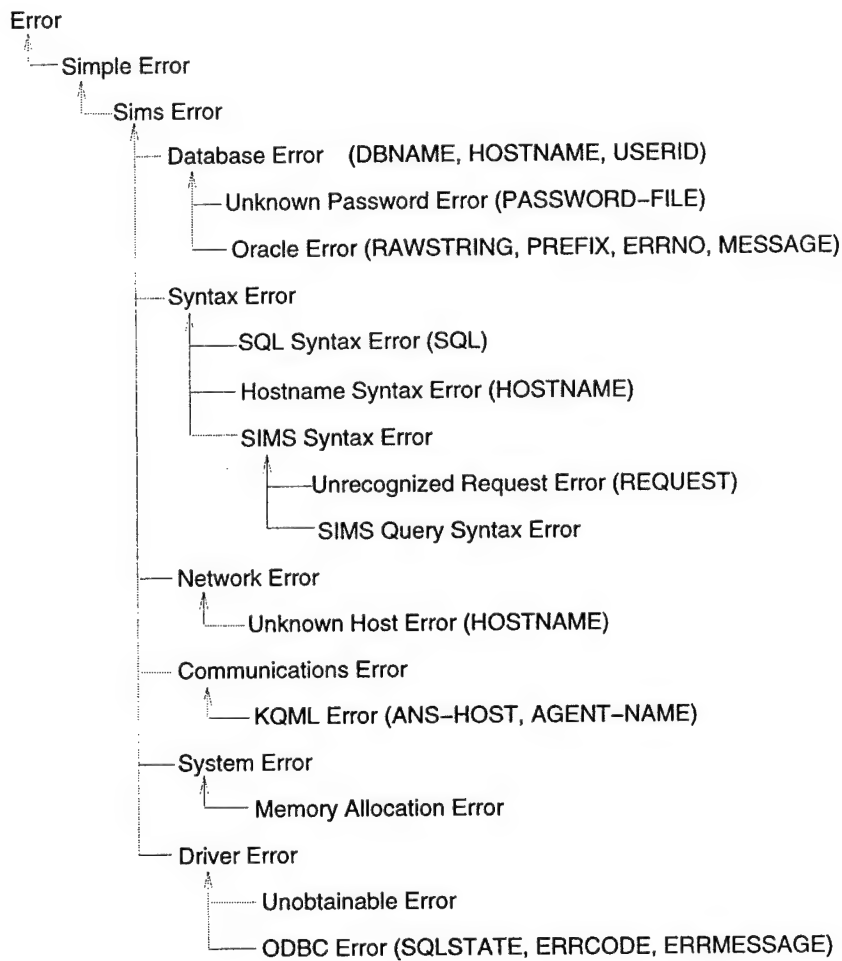   └── ODBC Error (SQLSTATE, ERRCODE, ERRMESSAGE)

Figure 18: SIMS Error Hierarchy

8 Installation and System Requirements

The SIMS system currently runs in Allegro Common Lisp 4.3 under Solaris on SUN workstations. Older versions which were released on ACL using other versions of Unix, on MCL 2.0 on the Macintosh and Lucid Common Lisp 4.0 on Unix are no longer fully supported. SIMS requires the following software components:

Loom provides the underlying knowledge representation and programming support. Loom is available by signing a license with USC/ISI. For more information, see `http://www.isi.edu/isd/LOOM/≈ LOOM-HOME.html` Currently, we are using Loom version 3.0, patch version 38 or greater.

MK-DEFSYSTEM is used to organize the Lisp files and providing system building and maintenance support. MK-DEFSYSTEM is currently included with Loom, and is available from the CMU AI Repository as well if needed (e.g., at `http://www-cgi.cs.cmu.edu/afs/cs.cmu.edu/project/≈ ai-repository/ai/lang/lisp/code/tools/defsys/mkant/defsys25.tgz`). We require version v2.5.

KQML (Knowledge Query and Manipulation Language) provides remote communication support between SIMS and remote DB servers, and between other information integration agents and SIMS. It can also be used to communicate between multiple SIMS servers. We are currently using KQML version 2.06. Included with the SIMS release are several patches which improve the performance of KQML in the SIMS world. KQML is available by signing a license with University of Maryland, Baltimore County. For more information, see `http://www.cs.umbc.edu/kqml/`.

ODBC. We include with SIMS code which provides a programmatic interface to Oracle databases using ODBC. We currently use ODBC 2.0 and are porting to ODBC 3.0; these products are from InterSolv and are available from third-party retailers. This component is also optional since SIMS can be run with any database wrapper you may choose to implement.

CORBA. We also include with SIMS code which interfaces between CORBA and KQML. This interface is entirely optional as well. It does require ORBIX version 2.2 from Iona Technologies to run. For more information on ORBIX, see `http://www.iona.com/`.

CVS. We use CVS to manage the software configuration of the SIMS system under Unix. Again, this is entirely optional. While using SIMS release does not require that you use CVS, it may be helpful for tracking your local development of source descriptions and domain models. CVS is GPL software and is available from `prep.ai.mit.edu` and other FSF FTP locations.

8.1 Component Structure

Here is our current component and directory structure, which we recommend that users adopt:

- ▷ `config` – configuration and customization of the SIMS release
- ▷ `documentation` – for any general-purpose documentation
- ▷ `domains` – for domain models, source descriptions, KB data (if relevant), and queries
    - ▷ *<domain-name>* – e.g., transportation, logistics
        - ▷ `domain` – for domain models and domain-level queries
        - ▷ `sources` – for source descriptions used in *<domain-name>*
        - ▷ *<source-name>* – e.g., airports-db, inventory-kb
- ▷ `sys` – for SIMS system code
    - ▷ `defsys` – MK-DEFSYSTEM files for various systems and modules making up SIMS
    - ▷ `modeling` – implements the information source ontology used by SIMS and the APIs for the various source types.

- ▷ `Planner` – Implementations of SIMS planner(s) including operators and source selection code.
- ▷ `Wrapper` – General code used for implementing information source wrappers (e.g., for WWW pages)
- ▷ `netutils` – code for interacting with WWW, sockets, and GUIs, and query logging, etc.

- ▷ `tools` – for self-contained auxiliary modules and third-party code used to support SIMS

    - ▷ `cl-http` – Common Lisp Hypermedia Server code
    - ▷ `corba2kqml` – Corba/KQML gateway
    - ▷ `lisp` – various lisp utils including multiprocessing, regular expression handling, and error reporting

8.2 Define, Load and Compile Components

As mentioned above, we use the MK-DEFSYSTEM from CMU (this comes with LOOM) to define each subsystem. Each software component and domain has its own system declaration file, e.g., `planner.system`, `example.system`. If you want to define your own subsystem, please see the files in the `defsys` directory for examples.

One can define a component that includes many other components. For example, there is a subsystem called `riscsims.system` which includes `common`, `planner`, `operators`, `preprocessor`, and `eval`.

Before you can use the defsystems, you will need to set two global variables. The first variable, `user::*ariadne-root-dir*`, sets the directory for the location of all of the subsystems:

`(setq *ariadne-root-dir* "/home/johndoe/sims/sys/ariadne/")`

The second variable, `mk::*central-registry*`, sets the location of the defsystem definitions for each of the components:

`(setq mk::*central-registry* '("/home/sims/sys/ariadne/sys/defsys/"))`

One can load a system using the command `mk:operate-on-system`. For example, to load `riscsims`, you do:

`(mk:operate-on-system :riscsims :load)`

You can substitute the keyword `:load` by `:compile` to compile the system.

`(mk:operate-on-system :riscsims :compile)`

You can also force the system to recompile all of the files in a component by appending the `:force` keyword:

`(mk:operate-on-system :riscsims :compile :force t)`

The typical sequence of loading SIMS is to load the `riscsims` system first, then load the optional components that you need, and finally load the domain model and source description information that are specific for your application.

For example, after loading in the riscsims system, you would load the example from the manual as follows:

`(mk:operate-on-system :example :load :compile-during-load t)`

8.3 Complete build procedure

Here are the steps to build a complete version of SIMS which will be able to execute the example queries shown in this manual by use of several sample knowledge base sources.

1. **Install and configure SIMS**

    (a) Obtain the file `SIMS.tar.gz`. This compressed archive is for Sun/Solaris and anticipates that you have Allegro CL 4.3 and Loom already installed at your site. Before you can use this archive, you should have already executed licenses for Loom and for KQML.

    (b) Use gunzip to uncompress that file into a regular tar file.

(c) Choose a directory into which to expand the tar archive. This will be your "root directory." Change to this directory.

(d) Use `tar -xf SIMS.tar .` to recreate the subdirectory structure in that directory.

(e) In your shell and/or `.cshrc` file, set the following variables. You may find it helpful to refer to the sample `.cshrc` file supplied in the `./config` subdirectory of the recreated archive.

    i. `setenv ARIADNE_ROOT` *root_directory*

    ii. `setenv KQML_HOME $ARIADNE_ROOT/tools/kqml/kqml-2.06/`

2. **Configure and build KQML**

(a) Change directory to KQML: `cd $KQML_HOME/src/lisp`

(b) `make -f Makefile.solaris tcp.so`

3. **Build basic SIMS in Lisp**

(a) Begin a Lisp session using your pre-built image of ACL 4.3. For best results, use an ACL image which uses shared libraries.

(b) Load Loom into the Lisp session. A typical way to do this is to load the file `load-loom.lisp` from the Loom 3.0 distribution directory. If the Lisp image you loaded in step 3a above includes Loom, you should omit this step. After loading Loom, you may want to checkpoint this image to disk for later reuse (e.g., using `(excl::dumplisp "my-checkpoint-loom-image")`).

(c) In your Lisp session and/or lisp init (typically, `~/.clinit.cl`) files, set the following variables, substituting for *ariadne_root* the physical pathname specified above for the environment variable `$ARIADNE_ROOT`. You may refer to the supplied file `$ARIADNE_ROOT/config/.clinit.cl` for an example of how these variables might be set.

    i. `(setq excl::*fasl-default-type* '"sfasl")`

    ii. `(setq mk::*filename-extensions* '("lisp" . "sfasl"))`

    iii. `(setq user::*ariadne-root-dir* '"`*ariadne_root*`")`

    iv. `(setq mk::*central-registry* '("`*ariadne_root*`/sys/defsys/"))`

(d) Build the Lisp RISCSIMS system.
`(mk::operate-on-system :riscsims :load :compile-during-load t)`

(e) At this point, the session contains a complete but "empty" version of SIMS; you may want to checkpoint this image to disk for later reuse.

4. **Add SIMS GUI [optional]**
To add in the Graphical User Interface written in Java:

(a) In a Unix shell, make sure your `CLASSPATH` environment variable contains both the standard class location as well as `$ARIADNE_ROOT/sys/netutils/gui/javagui/java/`. Also be sure that the `java` and `javac` executables are in your `PATH`.

(b) Change to the java directory. `cd $ARIADNE_ROOT/sys/netutils/gui/javagui/java/`.

(c) Execute `make` to compile the java classes.

(d) Load the GUI driver into the Lisp session.
`(mk::operate-on-system :javagui :load :compile-during-load t)`

(e) Finally, you may want to checkpoint this image as well.

5. **Add domain model(s) and source specifications(s).**

(a) For example, to add in the example domain, execute:
`(mk::operate-on-system :example :load :compile-during-load t)`

94

(b) After loading you may want to checkpoint this image as well.

6. **Customize and run SIMS**

SIMS is now loaded. To begin your session:

(a) Execute (`in-package :sims`) to select the SIMS package (namespace)

(b) Execute (`in-context :example`) to select the Example domain context.

At this point, the commands in section 6 should work. In particular, we recommend executing the commands in the following order:

(c) Load Lisp init file such as ~/`.clinit.cl` if this is not autoloaded.

(d) Set necessary SIMS configuration variables such as `kqml::*kqml-ans-host*` and etc. if they are not set by the init file.

(e) You may also set at this time important SIMS customization variables such as `ucpop::*close-broken-sources*`, `sims::*parallel-execution*`, `ucpop::*interleaved-execution*`, and `ucpop::*search-limit*`. Also execute one of (`sims::sims-trap-on` or (`sims::sims-trap-off`) and one of (`sims::sims-trace-on`) or (`sims::sims-trace-off`).

(f) Initialize information sources, using (`sims::initialize-all-sources`) or (`sims::initialize-source`).

(g) Execute (`sims::compile-axioms`) to build the network of source representations used by the planner.

(h) If you have loaded the GUI, you may start it by executing (`sims::sims-java-server`) followed a few seconds later by (`sims::java-interface`).

Examples of how one might carry out steps 6d- 6h can be found in the supplied file `$ARIADNE_ROOT/config/sims-init.lisp`.

(i) Plan and execute queries.

(j) To exit SIMS including severing any KQML connections, type (`sims::exit`).

7. **Connect SIMS to a relational DB source using ODBC [Optional]**

You may wish to skip this section on first pass. To extend the above with a database source such as the sample database EXDB included in the release, you will need to continue with the following:

(a) **Bring up database**

Install the database instance, if necessary, into Oracle or other RDBMS. For more information about bringing up a new database, please consult your RDBMS documentation.

This example will assume an Oracle DB is being used. Suppose the Oracle SID for this database is called `mydb` and it is running on `myhost.mycompany.com`. Suppose further that the ODBC libraries are installed in `/opt/odbc/odbc3.0/`.

(b) **Launch KQML ANS**

Verify that KQML is installed correctly and that the KQML Agent Name Service (ANS) is running on some host. Suppose we call this host `anshost.mycompany.com`.

Briefly, to start an ans on host `anshost.mycompany.com`:

  i. In a shell on `anshost.mycompany.com`
  ii. `cd $KQML_HOME/bin`
  iii. `source kqmlenv.csh`
  iv. `startans anshost`. KQML allows only one ANS per machine. By convention we name the ANS using the hostname. KQML should respond "Facilitator started."
  v. If it responds otherwise, you will have to kill that process using `kill -9` and try again.
  vi. To list agents registered with the ANS, do `agentls`. You should get back a listing such as `anshost (tcp-ip anshost.mycompany.com:5500:)`

95

For more information on KQML and the ANS, see the files `$KQML_HOME/doc/*.{html,doc,rtf,ps}`.

(c) **Configure ODBC**

In the file which is the value of the environment variable `ODBCINI`, in the [ODBC Data Sources] section make an entry for this database of the form:

```
myhost.mycompany.com:mydb=INTERSOLV Oracle V7 ODBC Driver
```

and in the Driver Configuration Section make an entry of the form:

```
[myhost.mycompany.com:mydb]
Driver=/opt/odbc/odbc3.0/lib/ivor712.so
Description=INTERSOLV Oracle Version 7 ODBC Driver
ServerName=T:myhost.mycompany.com:mydb
```

See the supplied file `$ARIADNE_ROOT/config/odbc.ini` for an example of how the file could be configured. The convention is that the ODBC aliases have the format *full_internet_hostname:database_SID*. This convention is used by the sql_server and SIMS to effectively communicate password information.

(d) **Build and configure the SQL server**

   i. Change to the `sql_server` directory:
   `cd $ARIADNE_ROOT/domains/common/sources/common/sql_server`

   ii. Edit the file `init.csh` in that directory to supply the correct values for the variables `LD_LIBRARY_PATH`, `ODBCHOME`, `IV_GLS_LCDIR`, `IV_GLS_REGISTRY`, `INFORMIXDIR`, `ODBCINI`, `KQML_HOME`, `KQML_ROUTER`, `KQML_ANS`, and `ORACLE_HOME`.

   iii. In the Unix shell, `source init.csh`. Check that the established environment variable values make sense.

   iv. Edit `Makefile` in that directory if needed, then use `make` in that directory. The result should be an executable called `server`.

   v. Start `server` with two arguments:
      - KQML agent name. By convention, this should be specified as `sql_server_for_myhost_mycompany_com`.
      - Password file to consult (see below).

      Note that this server will be able to communicate with *all* database instances running on `myhost.mycompany.com`.

   vi. Use `$KQML_HOME/bin/agentls` on `anshost.mycompany.com` to verify that the `sql_server` agent just started has registered correctly.

(e) **Configure SIMS to access the new source**

   i. Create or update the source file(s) for the example domain to reflect this newly configured source. In this case you will want to edit a copy of `$ARIADNE_ROOT/domains/example/≈ sources/exdb/definition.lisp`, ideally placed into a domain and source subdirectory of your own creation (see Section 8.1).

   ii. Create or update a password file which will map from SID/userid pairs to database passwords. In this file, create an entry of the form:

   ```
   # Example database
   mydb:myhost.mycompany.com:myuserid:mypasswd # SIMS release DB
   ```

   See the supplied file `$ARIADNE_ROOT/config/passwd` for an example of how this file might be configured.

   iii. In your Lisp session and/or lisp init (typically, `~/.clinit.cl`) file, set the following variable to point to the location of the above password file:

   A. `(setq sims::*password-file* <password_file_location>)`

(f) **Re-run SIMS**

At this point, the command sequence in Step 6 would be a good place to consult again.

96

9 Coded Example

This section gives the code that implements the example discussed throughout the manual.

```lisp
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; domain-model.lisp
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;; Domain model for the example in the SIMS user manual
;;;

(in-package :sims)
(in-context :example)

;;; Seaport relations

(def-sims-relation geoloc-code
    :domain seaport
    :range string)

(def-sims-relation port-name
    :domain seaport
    :range string)

(def-sims-relation cranes
    :domain seaport
    :range number)

(def-sims-relation depth
    :domain seaport
    :range number)

(def-sims-relation tariff-code
    :domain european-large-seaport
    :range string)

(def-sims-relation seaport-country-code
    :domain seaport
    :range string)

;;; Seaport concepts

(def-sims-concept american-large-seaport
    :is-primitive large-seaport
    :annotations ((key (geoloc-code))
                  (key (port-name))))

(def-sims-concept european-large-seaport
    :is-primitive (:and large-seaport
                        (:the tariff-code string))
    :annotations ((key (geoloc-code))
                  (key (port-name))))

(def-sims-concept large-seaport
    :is (:and seaport
             (> cranes 7))
    :annotations ((key (geoloc-code))
                  (key (port-name))
                  (covering (american-large-seaport
                             european-large-seaport))))

(def-sims-concept small-seaport
    :is (:and seaport
             (<= cranes 7))
    :annotations ((key (geoloc-code))
                  (key (port-name))))

(def-sims-concept seaport
```

```
    :is-primitive (:and sims-domain-concept
                        (:the country-of country)
                        (:the geoloc-code string)
                        (:the seaport-country-code string)
                        (:the port-name string)
                        (:the cranes number)
                        (:the depth number))
    :annotations ((key (geoloc-code))
                  (key (port-name))
                  (covering (large-seaport small-seaport))))

;;; Country relations

(def-sims-relation country-code
    :domain country
    :range string)

(def-sims-relation country-name
    :domain country
    :range string)

(def-sims-relation currency
    :domain country
    :range number)

(def-sims-relation language
    :domain country
    :range number)

;;; Country concepts

(def-sims-concept country
    :is-primitive (:and sims-domain-concept
                        (:the country-code string)
                        (:the country-name string)
                        (:the language string)
                        (:the currency string))
    :annotations ((key (country-code))))

(def-sims-relation country-of
    :domain seaport
    :range country
    :is (:satisfies (?s ?c)
          (:for-some (?country-code)
            (:and (seaport ?s)
                  (country ?c)
                  (seaport-country-code ?s ?country-code)
                  (country-code ?c ?country-code)))))
```

```lisp
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; queries.lisp
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;; Example queries
;;;

(in-package :sims)
(in-context :example)

(setq *queries*
  '(;; Large Seaport queries

    (11 "What the geoloc codes of all large seaports"
     (sims-retrieve (?geoloc-code)
       (:and (large-seaport ?ls)
         (geoloc-code ?ls ?geoloc-code))))

    (12 "How many cranes are available in various large seaports"
     (sims-retrieve (?cr)
       (:and (large-seaport ?ls)
         (cranes ?ls ?cr))))

    (13 "List the geoloc-codes and number of cranes for all large
seaports"
     (sims-retrieve (?geoloc-code ?cr)
       (:and (large-seaport ?ls)
         (geoloc-code ?ls ?geoloc-code)
         (cranes ?ls ?cr))))


    ...

    (106 "List currency and language for all countries"
     (sims-retrieve (?cc ?cn ?currency ?lang)
       (:and (country ?c)
         (country-code ?c ?cc)
         (country-name ?c ?cn)
         (currency ?c ?currency)
         (language ?c ?lang))))

    (107 "List all countries' currency, language, and seaport information"
     (sims-retrieve (?cc ?cn ?currency ?lang ?cr ?geoloc-code ?port-name)
       (:and (country ?c)
         (country-code ?c ?cc)
         (country-name ?c ?cn)
         (currency ?c ?currency)
         (language ?c ?lang)
         (seaport ?s)
         (cranes ?s ?cr)
         (geoloc-code ?s ?geoloc-code)
         (port-name ?s ?port-name)
         (country-of ?s ?c))))

    ))
```

99

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; exkb1.lisp
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;; Example source model of EXKB1 for SIMS manual
;;;

(in-package :sims)
(in-context :example)

;;; Define a new Loom KB data source called EXKB1

(define-source EXKB1 loom-kb-source)

;;; Describe the domain in terms of EXKB1

(source-description seaport s1 EXKB1
  (geoloc-code gc)
  (port-name pn)
  (cranes cr)
  (seaport-country-code scc))

(source-description country c1 EXKB1
  (country-code cc)
  (country-name cn)
  (language lang)
  (currency curr))

;;; Load data (facts) into this new data source

;;; Seaport data

(deffact EXKB1 S1 ABIDJAN
  (PN "Abidjan")
  (GC "AAPV")
  (CR 5)
  (SCC "IV"))
...
(deffact EXKB1 S1 VLORE
  (PN "Vlore")
  (GC "YALP")
  (CR 1)
  (SCC "AL"))

;;; Country data

(deffact EXKB1 C1 ALBANIA
  (CC "AL")
  (CN "ALBANIA")
  (LANG "ALBANIAN")
  (CURR "LEK"))
...
(deffact EXKB1 C1 ZIMBABWE
  (CC "ZI")
  (CN "ZIMBABWE")
  (LANG "ENGLISH")
  (CURR "DOLLAR"))
```

```lisp
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; exdb.lisp
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;; Example source model of EXDB for SIMS manual
;;;

(in-package :sims)
(in-context :example)

;;; Define a new SQL database, addressed using ODBC alias name,
;;; communication via KQML, called EXDB

(define-source EXDB kqml-odbc-sql-source
  :host "isd18.isi.edu"
  :agent-name "sql_server"
  :db-name "examplei"
  :userid "ifd")

;;; Describe the domain in terms of EXDB

(source-description large-seaport lgsp exdb
  (geoloc-code gc)
  (depth dp)
  (port-name pn)
  (cranes cr)
  (seaport-country-code scc))

(source-description small-seaport smsp exdb
  (geoloc-code gc)
  (port-name pn)
  (cranes cr)
  (seaport-country-code scc))

(source-description european-large-seaport lgeurosp exdb
  (geoloc-code gc)
  (depth dp)
  (port-name pn)
  (cranes cr)
  (tariff-code tc)
  (seaport-country-code scc))

(source-description american-large-seaport lgamersp exdb
  (geoloc-code gc)
  (depth dp)
  (port-name pn)
  (cranes cr)
  (seaport-country-code scc))

(source-description seaport sp exdb
  (geoloc-code gc)
  (port-name pn)
  (cranes cr)
  (seaport-country-code scc))

(source-description country ctry exdb
  (country-code cc)
  (country-name cn)
  (language lang)
  (currency curr))

;;; No KB facts
```

101

10 Additional Reading

Using this manual and following the instructions in it require familiarity with SIMS, as well as with the Loom knowledge representation language, and the KQML transport protocol.

The following papers may be consulted for further information about these programs.

10.1 SIMS

1. Ambite, J.L. and Craig A. Knoblock Reconciling Distributed Information Sources. *Working Notes of the AAAI Spring Symposium on Information Gathering in Distributed Heterogeneous Environments*, Palo Alto, CA, 1995.

2. Ambite, J.L., Yigal Arens, Naveen Ashish, Chin Y. Chee, Chun-Nan Hsu, Craig A. Knoblock Wei-Min Shen, and Sheila Tejada. 1995. *The SIMS Manual*, Version 1.0. ISI/TM-95-428.

3. Arens, Y., Craig A. Knoblock and Chun-Nan Hsu. Query Processing in the SIMS Information Mediator. *Advanced Planning Technology*, editor, Austin Tate, AAAI Press, Menlo Park, CA, 1996.

4. Arens, Y., Chee, C.Y., Hsu, C-N., and Knoblock, C.A. 1993. Retrieving and Integrating Data from Multiple Information Sources. In *International Journal of Intelligent and Cooperative Information Systems*. Vol. 2, No. 2. Pp. 127-158.

5. Arens, Y., Knoblock, C.A., and Shen W-M. Query Reformulation for Dynamic Information Integration, *Journal of Intelligent Information Systems*, 6(2/3):99-130, 1996.

6. Arens, Y. and Knoblock, C.A. 1994. Intelligent Caching: Selecting, Representing, and Reusing Data in an Information Server. In *Proceedings of the Third International Conference on Information and Knowledge Management (CIKM-94)*, Gaithersburg, MD.

7. Arens, Y., Chin Y. Chee, Chun-Nan Hsu, and Craig A. Knoblock Retrieving and Integrating Data from Multiple Information Sources. International Journal of Intelligent and Cooperative Information Systems. Vol. 2, No. 2. Pp. 127-158, 1993.

8. Arens, Y. and Knoblock, C.A. 1992. Planning and Reformulating Queries for Semantically-Modeled Multidatabase Systems, *Proceedings of the First International Conference on Information and Knowledge Management (CIKM-92)*, Baltimore, MD.

9. Hsu, C-N., and Knoblock, C.A. 1995. Estimating the Robustness of Discovered Knowledge, in *Proceedings of the First International Conference on Knowledge Discovery and Data Mining (KDD-95)*, Montreal, Quebec, Canada.

10. Hsu, C-N., and Knoblock, C.A. 1995. Using inductive learning to gen- erate rules for semantic query optimization. In Gregory Piatetsky-Shapiro and Usama Fayyad, editors, *Advances in Knowledge Discovery and Data Mining*, chapter 17. MIT Press.

11. Hsu, C-N., and Knoblock, C.A. 1994. Rule Induction for Semantic Query Optimization, in *Proceedings of the Eleventh International Conference on Machine Learning (ML-95)*, New Brunswick, NJ.

12. Hsu, C-N., and Knoblock, C.A. 1993. Reformulating Query Plans For Multidatabase Systems. In *Proceedings of the Second International Conference of Information and Knowledge Management (CIKM-93)*, Washington, D.C.

13. Hsu, C.-N. and Knoblock, C. A. Discovering Robust Knowledge from Dynamic Closed-World Data. *Proceedings of the Thirteenth National Conference on Artificial Intelligence*, Portland, Oregon, 1996.

14. Knoblock, C.A., Arens, Y. and Hsu, C-N. 1994. An Architecture for Information Retrieval Agents. In *Proceedings of the Second International Conference on Cooperative Information Systems*, University of Toronto Publications, Toronto, Ontario, Canada.

15. Knoblock, C.A. 1995. Planning, Executing, Sensing, and Replanning for Information Gathering. In *IJCAI-95*, Montreal, Quebec, Canada.

16. Craig A. Knoblock Applying a General-Purpose Planner to the Problem of Query Access Planning. *Proceedings of the AAAI Fall Symposium on Planning and Learning: On to Real Applications*, 1994.

17. Knoblock, C.A. 1994. Generating Parallel Execution Plans with a Partial-Order Planner. *Artificial Intelligence Planning Systems: Proceedings of the Second International Conference (AIPS94)*, Chicago, IL.

18. Craig A. Knoblock Building a Planner for Information Gathering: A Report from the Trenches Artificial Intelligence Planning Systems: *Proceedings of the Third International Conference* (AIPS96), Edinburgh, Scotland, 1996.

19. Craig A. Knoblock and Jose Luis Ambite. Agents for Information Gathering *Software Agents*, J. Bradshaw ed., AAAI/MIT Press, Menlo Park, CA, 1997.

20. Craig A. Knoblock, Yigal Arens, and Chun-Nan Hsu. Cooperating Agents for Information Retrieval. *Proceedings of the Second International Conference on Cooperative Information Systems*, Toronto, Ontario, Canada, University of Toronto Press, 1994.

These publications, as well as additional information about SIMS, can be accessed through the WWW at `http://www.isi.edu/sims/`.

10.2 Loom

1. MacGregor, R. A Deductive Pattern Matcher. In *Proceedings of AAAI-88, The National Conference on Artificial Intelligence*. St. Paul, MN, August 1988.

2. MacGregor, R. The Evolving Technology of Classification-Based Knowledge Representation Systems. In John Sowa (ed.), *Principles of Semantic Networks: Explorations in the Representation of Knowledge*. Morgan Kaufmann. 1990.

Additional papers and information about Loom can be accessed trough the WWW at the Loom Project homepage: `http://www.isi.edu/isd/LOOM/LOOM-HOME.html` .

10.3 KQML

1. Finin, T., Fritzson, R. and McKay, D. A Language and Protocol to Support Intelligent Agent Interoperability. In *Proceedings of the CE and CALS Washington '92 Conference*, June, 1992.

Additional papers and information about KQML can be accessed through the WWW at the KQML homepage: `http://www.cs.umbc.edu/kqml/` .

10.4 CORBA related

1. Iona Technologies. Orbix 2.2: Programming Guide. March 1997.

2. Object Management group. The Common Object Request Broker: architecture and specification. OMG Document Number 91.12.1, 1991.

3. Ion Muslea. A Guide for Making SIMS a CORBA-compliant Application

Acknowledgements

References

[1] R.J. Brachman and J.G. Schmolze. An overview of the KL-ONE knowledge representation system. *Cognitive Science*, 9(2):171–216, 1985.

[2] Tim Finin, Rich Fritzson, and Don McKay. A language and protocol to support intelligent agent interoperability. In *Proceedings of the CE and CALS*, Washington, D.C., June 1992.

[3] Object Management group. *The Common Object Request Broker: architecture and specification*, volume OMG Document Number 91.12.1. Object Management group, 1991.

[4] Robert MacGregor. A deductive pattern matcher. In *Proceedings of the Seventh National Conference on Artificial Intelligence*, Saint Paul, Minnesota, 1988.

[5] Robert MacGregor. The evolving technology of classification-based knowledge representation systems. In John Sowa, editor, *Principles of Semantic Networks: Explorations in the Representation of Knowledge*. Morgan Kaufmann, 1990.

[6] Ion Muslea. A guide for making sims a corba-compliant application. Technical Report ISI, USC, 1996.

[7] Iona Technologies. *Orbix 2.2: Programming Guide*. Iona Technologies, March 1997.

DISTRIBUTION LIST

addresses	number of copies
RAYMOND A. LIUZZI AFRL/IFTB 525 BROOKS ROAD ROME NY 13441-4505	5
USC INFORMATION SCIENCES INSTITUTE 4676 ADMIRALTY WAY MARINA DEL REY CA 90292-6695	5
AFRL/IFOIL TECHNICAL LIBRARY 26 ELECTRONIC PKY ROME NY 13441-4514	1
ATTENTION: DTIC-OCC DEFENSE TECHNICAL INFO CENTER 8725 JOHN J. KINGMAN ROAD, STE 0944 FT. BELVOIR, VA 22060-6218	2
DEFENSE ADVANCED RESEARCH PROJECTS AGENCY 3701 NORTH FAIRFAX DRIVE ARLINGTON VA 22203-1714	1
RELIABILITY ANALYSIS CENTER 201 MILL ST. ROME NY 13440-8200	1
ATTN: GWEN NGUYEN GIDEP P.O. BOX 8000 CORONA CA 91718-8000	1
AFIT ACADEMIC LIBRARY/LDEE 2950 P STREET AREA B, BLDG 642 WRIGHT-PATTERSON AFB OH 45433-7765	1

```
ATTN:  GILBERT G. KUPERMAN                          1
AL/CFHI, BLDG. 248
2255 H STREET
WRIGHT-PATTERSON AFB OH 45433-7022


ATTN: TECHNICAL DOCUMENTS CENTER                    1
OL AL HSC/HRG
2698 G STREET
WRIGHT-PATTERSON AFB OH  45433-7604


AIR UNIVERSITY LIBRARY (AUL/LSAD)                   1
600 CHENNAULT CIRCLE
MAXWELL AFB AL 36112-6424


US ARMY SSDC                                        1
P.O. BOX 1500
ATTN: CSSD-IM-PA
HUNTSVILLE AL 35807-3801


TECHNICAL LIBRARY D0274(PL-TS)                      1
SPAWARSYSCEN
53560 HULL STREET
SAN DIEGO CA 92152-5001


NAVAL AIR WARFARE CENTER                            1
WEAPONS DIVISION
CODE 4BL000D
1 ADMINISTRATION CIRCLE
CHINA LAKE CA 93555-6100

SPACE & NAVAL WARFARE SYSTEMS CMD                   2
ATTN: PMW163-1 (R. SKIANO)RM 1044A
53560 HULL ST.
SAN DIEGO, CA  92152-5002


SPACE & NAVAL WARFARE SYSTEMS                       1
COMMAND, EXECUTIVE DIRECTOR (PD13A)
ATTN:  MR. CARL ANDRIANI
2451 CRYSTAL DRIVE
ARLINGTON VA 22245-5200

COMMANDER, SPACE & NAVAL WARFARE                    1
SYSTEMS COMMAND (CODE 32)
2451 CRYSTAL DRIVE
ARLINGTON VA 22245-5200
```

```
NSA/CSS                                                     1
K1
FT MEADE MD 20755-6000


ATTN: OM CHAUHAN                                            1
DCMC WICHITA
271 WEST THIRD STREET NORTH
SUITE 6000
WICHITA KS  67202-1212

AFRL/VSOS-TL (LIBRARY)                                      1
5 WRIGHT STREET
HANSCOM AFB MA 01731-3004


ATTN:  EILEEN LADUKE/D460                                   1
MITRE CORPORATION
202 BURLINGTON RD
BEDFORD MA 01730


OUSD(P)/DTSA/DUTD                                           2
ATTN:  PATRICK G. SULLIVAN, JR.
400 ARMY NAVY DRIVE
SUITE 300
ARLINGTON VA 22202

SOFTWARE ENGR'G INST TECH LIBRARY                          1
ATTN:  MR DENNIS SMITH
CARNEGIE MELLON UNIVERSITY
PITTSBURGH PA 15213-3890


USC-ISI                                                    1
ATTN:  DR ROBERT M. BALZER
4676 ADMIRALTY WAY
MARINA DEL REY CA 90292-6695


KESTREL INSTITUTE                                          1
ATTN:  DR CORDELL GREEN
1801 PAGE MILL ROAD
PALO ALTO CA 94304


ROCHESTER INSTITUTE OF TECHNOLOGY                          1
ATTN:  PROF J. A. LASKY
1 LOMB MEMORIAL DRIVE
P.O. BOX 9887
ROCHESTER NY 14613-5700
```

AFIT/ENG   1
ATTN:TOM HARTRUM
WPAFB OH 45433-6583

THE MITRE CORPORATION   1
ATTN: MR EDWARD H. BENSLEY
BURLINGTON RD/MAIL STOP A350
BEDFORD MA 01730

UNIV OF ILLINOIS, URBANA-CHAMPAIGN   1
ATTN: ANDREW CHIEN
DEPT OF COMPUTER SCIENCES
1304 W. SPRINGFIELD/240 DIGITAL LAB
URBANA IL 61801

HONEYWELL, INC.   1
ATTN: MR BERT HARRIS
FEDERAL SYSTEMS
7900 WESTPARK DRIVE
MCLEAN VA 22102

SOFTWARE ENGINEERING INSTITUTE   1
ATTN: MR WILLIAM E. HEFLEY
CARNEGIE-MELLON UNIVERSITY
SEI 2218
PITTSBURGH PA 15213-38990

UNIVERSITY OF SOUTHERN CALIFORNIA   1
ATTN: DR. YIGAL ARENS
INFORMATION SCIENCES INSTITUTE
4676 ADMIRALTY WAY/SUITE 1001
MARINA DEL REY CA 90292-6695

COLUMBIA UNIV/DEPT COMPUTER SCIENCE   1
ATTN: DR GAIL E. KAISER
450 COMPUTER SCIENCE BLDG
500 WEST 120TH STREET
NEW YORK NY 10027

SOFTWARE PRODUCTIVITY CONSORTIUM   1
ATTN: MR ROBERT LAI
2214 ROCK HILL ROAD
HERNDON VA 22070

AFIT/ENG   1
ATTN: DR GARY B. LAMONT
SCHOOL OF ENGINEERING
DEPT ELECTRICAL & COMPUTER ENGRG
WPAFB OH 45433-6583

```
NSA/OFC OF RESEARCH                                    1
ATTN:   MS MARY ANNE OVERMAN
9800 SAVAGE ROAD
FT GEORGE G. MEADE MD 20755-6000


AT&T BELL LABORATORIES                                 1
ATTN:   MR PETER G. SELFRIDGE
ROOM 3C-441
600 MOUNTAIN AVE
MURRAY HILL NJ 07974


ODYSSEY RESEARCH ASSOCIATES, INC.                      1
ATTN:   MS MAUREEN STILLMAN
301A HARRIS B. DATES DRIVE
ITHACA NY 14850-1313


TEXAS INSTRUMENTS INCORPORATED                         1
ATTN:   DR DAVID L. WELLS
P.O. BOX 655474, MS 238
DALLAS TX 75265


TEXAS A & M UNIVERSITY                                 1
ATTN:   DR PAULA MAYER
KNOWLEDGE BASED SYSTEMS LABORATORY
DEPT OF INDUSTRIAL ENGINEERING
COLLEGE STATION TX 77843


KESTREL DEVELOPMENT CORPORATION                        1
ATTN:   DR RICHARD JULLIG
3260 HILLVIEW AVENUE
PALO ALTO CA 94304


DARPA/ITO                                              1
ATTN:   DR KIRSTIE BELLMAN
3701 N FAIRFAX DRIVE
ARLINGTON VA 22203-1714


NASA/JOHNSON SPACE CENTER                              1
ATTN:   CHRIS CULBERT
MAIL CODE PT4
HOUSTON TX 77058


SAIC                                                   1
ATTN:   LANCE MILLER
MS T1-6-3
PO BOX 1303 (OR 1710 GOODRIDGE DR)
MCLEAN VA 22102
```

STERLING IMD INC.   1
KSC OPERATIONS
ATTN: MARK MAGINN
BEECHES TECHNICAL CAMPUS/RT 26 N.
ROME NY 13440

NAVAL POSTGRADUATE SCHOOL   1
ATTN: BALA RAMESH
CODE AS/RS
ADMINISTRATIVE SCIENCES DEPT
MONTEREY CA 93943

HUGHES SPACE & COMMUNICATIONS   1
ATTN: GERRY BARKSDALE
P. O. BOX 92919
BLDG R11 MS M352
LOS ANGELES, CA 90009-2919

SCHLUMBERGER LABORATORY FOR   1
   COMPUTER SCIENCE
ATTN: DR. GUILLERMO ARANGO
8311 NORTH FM620
AUSTIN, TX 78720

DECISION SYSTEMS DEPARTMENT   1
ATTN: PROF WALT SCACCHI
SCHOOL OF BUSINESS
UNIVERSITY OF SOUTHERN CALIFORNIA
LOS ANGELES, CA 90089-1421

SOUTHWEST RESEARCH INSTITUTE   1
ATTN: BRUCE REYNOLDS
6220 CULEBRA ROAD
SAN ANTONIO, TX 78228-0510

NATIONAL INSTITUTE OF STANDARDS   1
   AND TECHNOLOGY
ATTN: CHRIS DABROWSKI
ROOM A266, BLDG 225
GAITHSBURG MD 20899

EXPERT SYSTEMS LABORATORY   1
ATTN: STEVEN H. SCHWARTZ
NYNEX SCIENCE & TECHNOLOGY
500 WESTCHESTER AVENUE
WHITE PLAINS NY 20604

NAVAL TRAINING SYSTEMS CENTER   1
ATTN: ROBERT BREAUX/CODE 252
12350 RESEARCH PARKWAY
ORLANDO FL 32826-3224

```
CENTER FOR EXCELLENCE IN COMPUTER-              1
   AIDED SYSTEMS ENGINEERING
ATTN:   PERRY ALEXANDER
2291 IRVING HILL ROAD
LAWRENCE KS 66049


DR JOHN SALASIN                                 1
DARPA/ITO
3701 NORTH FAIRFAX DRIVE
ARLINGTON VA 22203-1714



DR BARRY BOEHM                                  1
DIR, USC CENTER FOR SW ENGINEERING
COMPUTER SCIENCE DEPT
UNIV OF SOUTHERN CALIFORNIA
LOS ANGELES CA 90089-0781


DR STEVE CROSS                                  1
CARNEGIE MELLON UNIVERSITY
SCHOOL OF COMPUTER SCIENCE
PITTSBURGH PA 15213-3891



DR MARK MAYBURY                                 1
MITRE CORPORATION
ADVANCED INFO SYS TECH; G041
BURLINTON ROAD, M/S K-329
BEDFORD MA 01730


ISX                                             1
ATTN:   MR. SCOTT FOUSE
4353 PARK TERRACE DRIVE
WESTLAKE VILLAGE,CA 91361


MR GARY EDWARDS                                 1
ISX
433 PARK TERRACE DRIVE
WESTLAKE VILLAGE CA 91361


DR ED WALKER                                    1
BBN SYSTEMS & TECH CORPORATION
10 MOULTON STREET
CAMBRIDGE MA 02238


LEE ERMAN                                       1
CIMFLEX TEKNOWLEDGE
1810 EMBACADERO ROAD
P.O. BOX 10119
PALO ALTO CA 94303
```

DR. DAVE GUNNING   1
DARPA/ISO
3701 NORTH FAIRFAX DRIVE
ARLINGTON VA   22203-1714

DAN WELD   1
UNIVERSITY OF WASHINGTON
DEPART OF COMPUTER SCIENCE & ENGIN
BOX 352350
SEATTLE, WA 98195-2350

STEPHEN SODERLAND   1
UNIVERSITY OF WASHINGTON
DEPT OF COMPUTER SCIENCE & ENGIN
BOX 352350
SEATTLE, WA 98195-2350

DR. MICHAEL PITTARELLI   1
COMPUTER SCIENCE DEPART
SUNY INST OF TECH AT UTICA/ROME
P.O. BOX 3050
UTICA, NY 13504-3050

CAPRARO TECHNOLOGIES, INC   1
ATTN: GERARD CAPRARO
311 TURNER ST.
UTICA, NY 13501

USC/ISI   1
ATTN: BOB MCGREGOR
4676 ADMIRALTY WAY
MARINA DEL REY, CA 90292

SRI INTERNATIONAL   1
ATTN: ENRIQUE RUSPINI
333 RAVENSWOOD AVE
MENLO PARK, CA 94025

DARTMOUTH COLLEGE   1
ATTN: DANIELA RUS
DEPT OF COMPUTER SCIENCE
11 ROPE FERRY ROAD
HANOVER, NH 03755-3510

UNIVERSITY OF FLORIDA   1
ATTN: ERIC HANSON
CISE DEPT 456 CSE
GAINESVILLE, FL 32611-6120

```
CARNEGIE MELLON UNIVERSITY                          1
ATTN:  TOM MITCHELL
COMPUTER SCIENCE DEPARTMENT
PITTSBURGH, PA 15213-3890


CARNEGIE MELLON UNIVERSITY                          1
ATTN:  MARK CRAVEN
COMPUTER SCIENCE DEPARTMENT
PITTSBURGH, PA 15213-3890


UNIVERSITY OF ROCHESTER                             1
ATTN:  JAMES ALLEN
DEPARTMENT OF COMPUTER SCIENCE
ROCHESTER, NY 14627


TEXTWISE, LLC                                       1
ATTN:  LIZ LIDDY
2-121 CENTER FOR SCIENCE & TECH
SYRACUSE, NY 13244


WRIGHT STATE UNIVERSITY                             1
ATTN:  DR. BRUCE BERRA
DEPART OF COMPUTER SCIENCE & ENGIN
DAYTON, OHIO 45435-0001


UNIVERSITY OF FLORIDA                               1
ATTN:  SHARMA CHAKRAVARTHY
COMPUTER & INFOR SCIENCE DEPART
GAINESVILLE, FL 32622-6125


KESTREL INSTITUTE                                   1
ATTN:  DAVID ESPINOSA
3260 HILLVIEW AVENUE
PALO ALTO, CA 94304


STOLLER-HENKE ASSOCIATES                            1
ATTN:  T.J. GOAN
2016 BELLE MONTI AVENUE
BELMONT, CA 94002


USC/INFORMATION SCIENCE INSTITUTE                   1
ATTN:  DR. CARL KESSELMAN
11474 ADMIRALTY WAY, SUITE 1001
MARINA DEL REY, CA 90292
```

MASSACHUSETTS INSTITUTE OF TECH   1
ATTN: DR. MICHAELE SIEGEL
SLOAN SCHOOL
77 MASSACHUSETTS AVENUE
CAMBRIDGE, MA 02139

USC/INFORMATION SCIENCE INSTITUTE   1
ATTN: DR. WILLIAM SWARTHOUT
11474 ADMIRALTY WAY, SUITE 1001
MARINA DEL REY, CA 90292

STANFORD UNIVERSITY   1
ATTN: DR. GIO WIEDERHOLD
857 SIERRA STREET
STANFORD
SANTA CLARA COUNTY, CA 94305-4125

NCCOSC RDTE DIV D44208   1
ATTN: LEAH WONG
53245 PATTERSON ROAD
SAN DIEGO, CA 92152-7151

SPAWAR SYSTEM CENTER   1
ATTN: LES ANDERSON
271 CATALINA BLVD, CODE 413
SAN DIEGO CA 92151

GEORGE MASON UNIVERSITY   1
ATTN: SUSHIL JAJODIA
ISSE DEPT
FAIRFAX, VA 22030-4444

DIRNSA   1
ATTN: MICHAEL R. WARE
DOD, NSA/CSS (R23)
FT. GEORGE G. MEADE MD 20755-6000

DR. JIM RICHARDSON   1
3660 TECHNOLOGY DRIVE
MINNEAPOLIS, MN 55418

LOUISIANA STATE UNIVERSITY   1
COMPUTER SCIENCE DEPT
ATTN: DR. PETER CHEN
257 COATES HALL
BATON ROUGE, LA 70803

INSTITUTE OF TECH DEPT OF COMP SCI   1
ATTN: DR. JAIDEEP SRIVASTAVA
4-192 EE/CS
200 UNION ST SE
MINNEAPOLIS, MN 55455

GTE/BBN   1
ATTN: MAURICE M. MCNEIL
9655 GRANITE RIDGE DRIVE
SUITE 245
SAN DIEGO, CA 92123

UNIVERSITY OF FLORIDA   1
ATTN: DR. SHARMA CHAKRAVARTHY
E470 CSE BUILDING
GAINESVILLE, FL 32611-6125

MISSION
OF
AFRL/INFORMATION DIRECTORATE (IF)

The advancement and application of information systems science and technology for aerospace command and control and its transition to air, space, and ground systems to meet customer needs in the areas of Global Awareness, Dynamic Planning and Execution, and Global Information Exchange is the focus of this AFRL organization. The directorate's areas of investigation include a broad spectrum of information and fusion, communication, collaborative environment and modeling and simulation, defensive information warfare, and intelligent information systems technologies.